RSARTE Git Integration

Anders Ek
IBM

RSARTE GIT INTEGRATION....1
INTRODUCTION....3
EGIT BRIEF OVERVIEW....3
GETTING STARTED....9
ECLIPSE PROJECTS AND GIT REPOSITORIES....9
ACCESSING A REMOTE GIT REPOSITORY....9
IMPORTING AN EXISTING REPOSITORY....11
STORING AN EXISTING PROJECT IN GIT....14
GIT STAGING – CHECKING WORK DONE AND COMMITTING THE RESULT....17
GIT STAGING VIEW....18
INVESTIGATING CHANGES....19
COMMITTING CHANGES....20
AMENDING CHANGES TO PREVIOUS COMMIT....20
GIT HISTORY – INVESTIGATING COMMITS AND BRANCHES....21
THE HISTORY VIEW....21
FILTERING THE HISTORY VIEW....22
SEARCHING THE HISTORY VIEW....23
COMPARING TWO UNRELATED COMMITS – GIT TREE COMPARE VIEW....24
USING GIT REPOSITORIES VIEW TO INVESTIGATE BRANCHES....25
CHECKING OUT AND CREATING BRANCHES....27
CHECK OUT A BRANCH....27
CREATE A NEW BRANCH....28
USING GIT REPOSITORIES VIEW TO CHECK OUT AND CREATE BRANCHES....30
MERGE AND REBASE....30
MERGE....31
REBASE....34
REWRITE HISTORY USING SQUASH MERGE....35
MERGING A SPECIFIC COMMIT USING CHERRY PICK....38
INTERACTIVE REBASE – ADVANCED REWRITING OF HISTORY....39
RESOLVING CONFLICTS....41
MERGE CONFLICTS....41
REBASE CONFLICTS....44
WORKING WITH REMOTE REPOSITORIES....47
REMOTE REPOSITORIES....48
REMOTE TRACKING BRANCHES....48
UPDATE FROM REMOTE REPOSITORIES: FETCH....49
UPSTREAM CONFIGURATIONS – LOCAL BRANCHES VS. REMOTE TRACKING BRANCHES....51

Update from Remote Repositories: Pull....52
Updating the Remote Repository with Your Local Changes using Push....53
COLLABORATING WITHIN A TEAM USING REMOTE BRANCHES....54
Creating a Remote Branch....54
Configuration of and Basic Operations for Remote Branches....55
WORKING WITH REMOTE REPOSITORIES USING GERRIT REVIEW SUPPORT....55
Cloning and Initiating a Repository from Gerrit....56
Gerrit Basics....57
Basic Operations when using a Gerrit Based Repository....58
Push for Review....58
Perform a Review using Mylyn....60
Installing and Configuring Mylyn....60
Reviewing a Change Request in Mylyn....64
Using Amend Commit to Fix Review Comments....67
OTHER COMMON SCENARIOS – STASHING, RESETTING AND TAGGING....69
Stashing – Temporarily Saving Your Changes....69
Reset – Reverting Your Changes....72
Tagging – Marking Important Commits....73
INTEGRATING RSARTE WITH COMMAND LINE GIT....75
Command Line RSARTE Invocation....75
Merge Integration....76
Git Hooks....77

This document describes how to handle models developed with Rational Software Architect Real-Time Edition (RSARTE) using Git version management.

The document was produced using RSARTE 9.1.2 CP1 and later updated for RSARTE 9.1.2 CP1 2015.34/37. All screen shots were captured on the Linux platform.

Introduction

Git is one of the most popular source code management systems used today. In this document we will look at how to use RSARTE together with Git. The focus is on model aspects and it assumes a basic knowledge of Git.
Good general purpose introductions to Git are for example:

- https://git-scm.com/book/en/v2/
  - A good general purpose on-line book about Git
- https://www.kernel.org/pub/software/scm/git/docs/gittutorial.html
  - A good getting started tutorial that describes basic work flows
- https://www.kernel.org/pub/software/scm/git/docs/user-manual.html
  - Git user manual

RSARTE can be used with Git in two different modes:

- Using the command line Git interface described by the links above
- Using the Eclipse integration (EGit)

EGit is a special purpose rewrite of Git in Java that enables convenient access to Git features from within the Eclipse/RSARTE graphical user interface. A good introduction to EGit is the EGit user guide available at http://wiki.eclipse.org/EGit/User_Guide.
The command line Git and the EGit Eclipse integration have different benefits and drawbacks:

- Command line Git is fast and efficient, but can be a daunting for new users. It also requires an environment specific configuration to work for model files.
- EGit is in most situations more convenient and easy to use than the command line interface since it is available directly in the RSARTE modeling environment but has slower performance than the command line implementation.

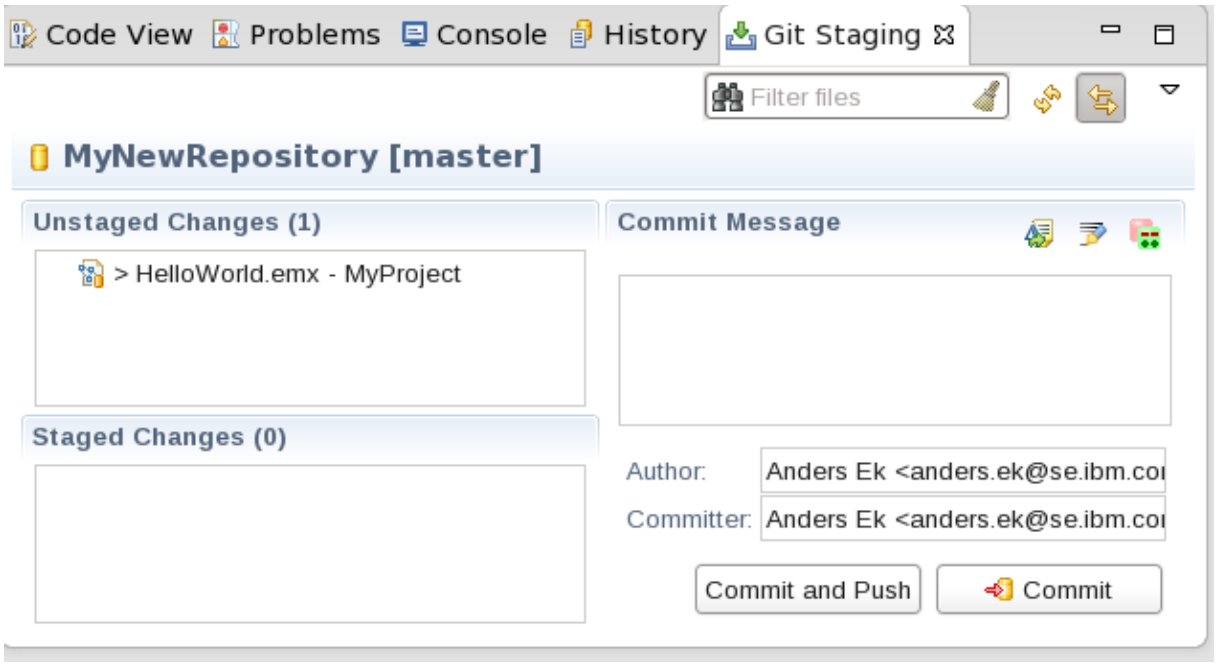
In this document we will look at both variants and it is also possible to use both in combination.

EGit Brief Overview

Before looking into different workflows in more detail we will take a brief tour of the most common EGit views you will use. There are three main views added or used by EGit:

- Git Staging View
- History View
- Git Repositories View

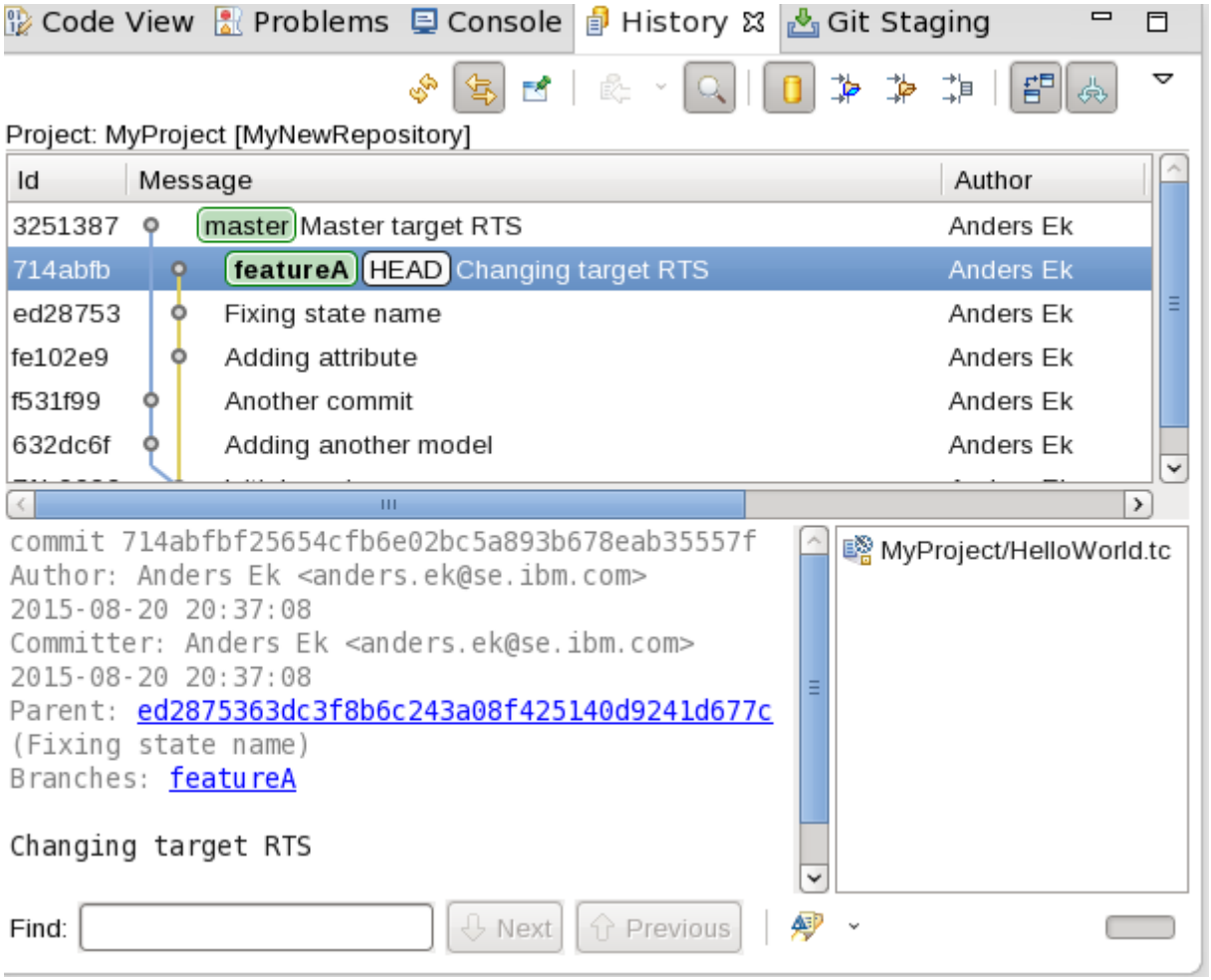
The first view you are likely to encounter is the Git Staging View:



You will use the Git Staging View to understand the state of your workspace, see what files have changed etc. and also to add your changes to the Git repository.

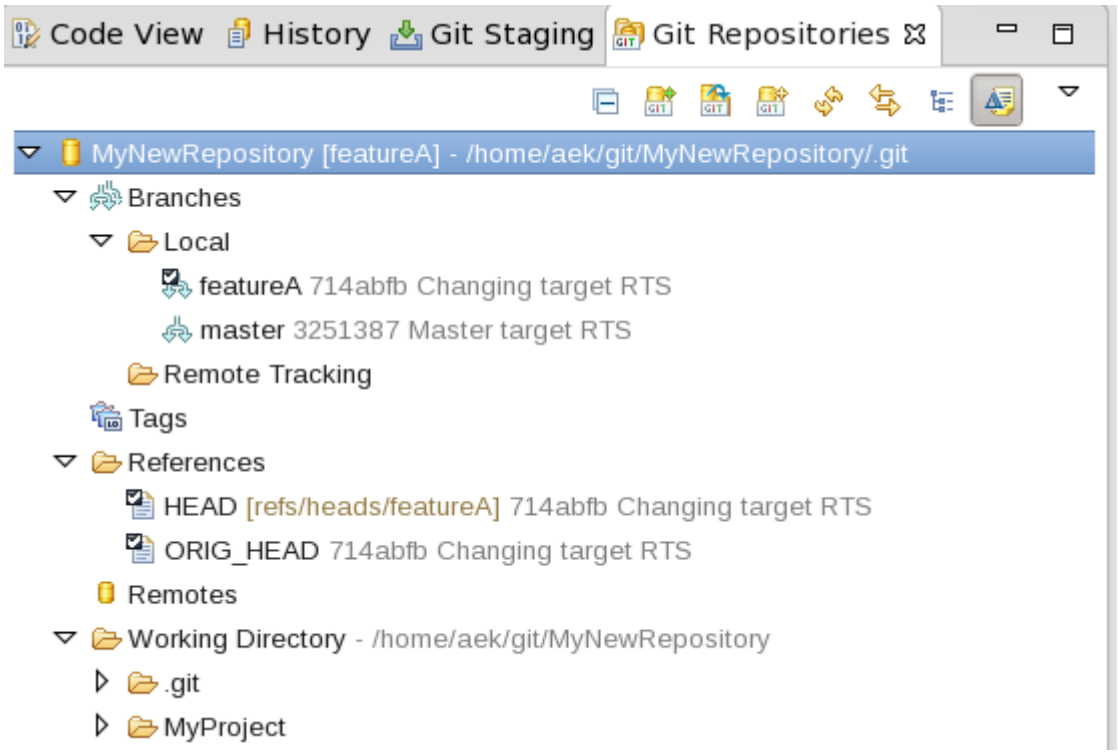
We will look in more detail into the different widgets and how to work with the view in section Git Staging – Checking work done and committing the result.

The History view is strictly speaking not added by EGit. It is used also for other source code management integrations, but EGit provides the contents when you use Git.



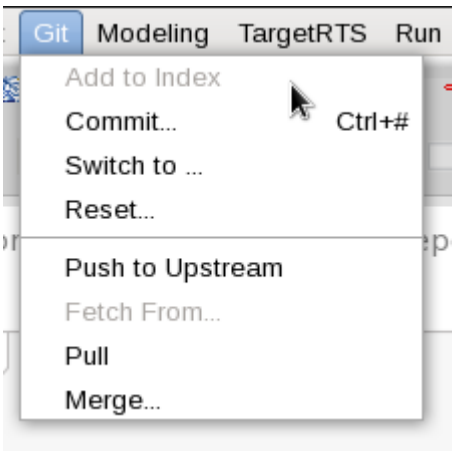
You will use the History view whenever you are interested in the previous actions done to your model. It provides information about who did what in a searchable graphical view sorted by date. We will look into this view more in section Git History – Investigating Commits and Branches

The third main view provided by EGit is the Git Repositories view.

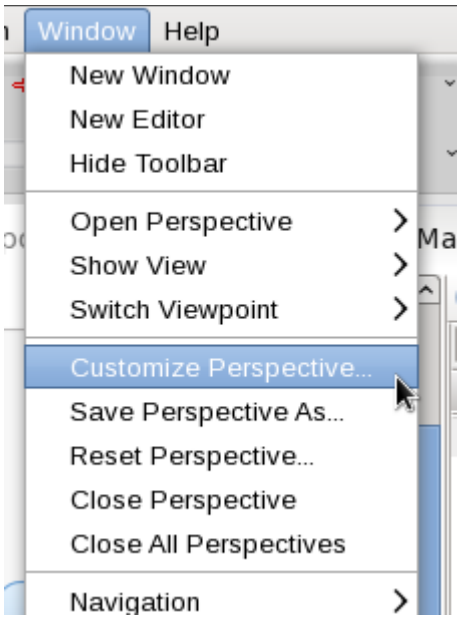


The Git Repositories view gives a summary of key aspects of the repositories you currently are working with inside RSARTE. It provides information about the branches that exist (including which branch you currently are working on), what remote repositories you have configured and where in the file system each repository is located. We will encounter the Git Repositories view in many contexts in this document, including in section Git History – Investigating Commits and Branches and in section Checking Out and Creating Branches.

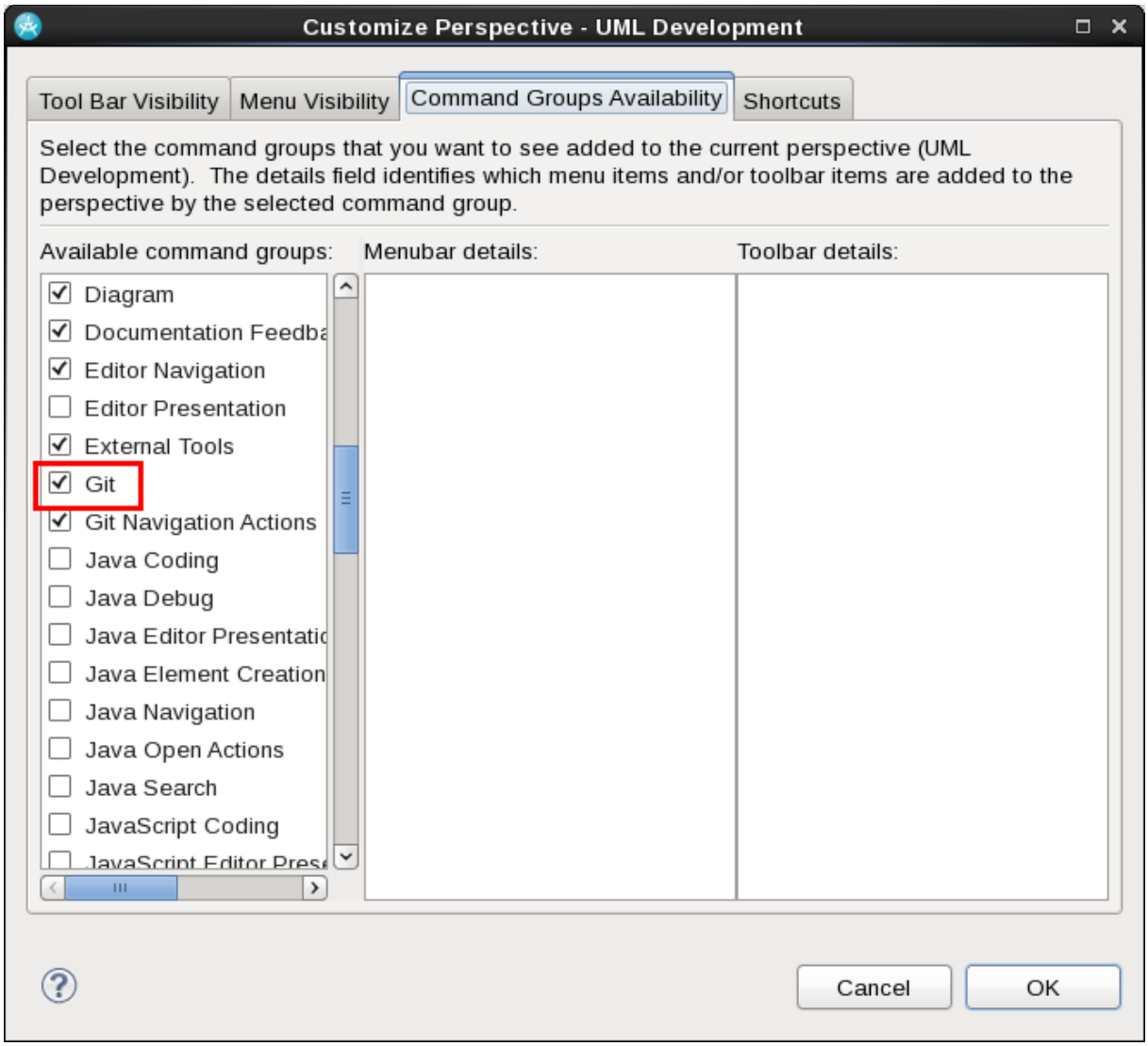
Git also provides one top-level menu and a toolbar.



To get the menu and toolbar buttons in the UML Development perspective recommended for RSARTE daily usage you need to enable them using the ”Customize Perspective” command available in the ”Window” menu.



Make sure to enable ”Git” in the ”Command Groups Availability” section.



The top-level menu and toolbar are strictly speaking not necessary, since all commands are available in the different Git views, but can in some situations be convenient.

In general you can see if you have EGit installed as part of RSARTE in a number of different ways. One very obvious way is that the Project Explorer by default annotates the Project Explorer with the repository and branch for all projects stored in Git.

Another difference when you have EGit installed is that when you do any Git action that will change the workspace contents, like for example when loading a new workspace or when

switching branches in Git, then EGit will update the status information. This may typically take a couple of seconds and you can see the status bar (at the bottom of the RSARTE window) indicating that something is happening. For example when checking out you might see the following.

Getting Started

Eclipse Projects and Git Repositories

Git is distributed version management system, that relies on a local repository storing information about the different versions of files and directories. In general a Git repository is a directory containing a copy of all files and folders plus a special top level directory called ".git". This directory contains the version information for all files and directories in the repository.
There are several different schemes available when combining Git with Eclipse projects, but in general it is recommended to have the ".git" repository directory outside of the Eclipse project structure. A typical structure is thus:

- MyApplication: The root folder of the repository.
  - .git: The Git version information directory
  - Project1: First Eclipse project
  - Project2: Second Eclipse project

A structure similar to this is expected by most commands in EGit and will improve the performance of the tools.

Accessing a Remote Git Repository

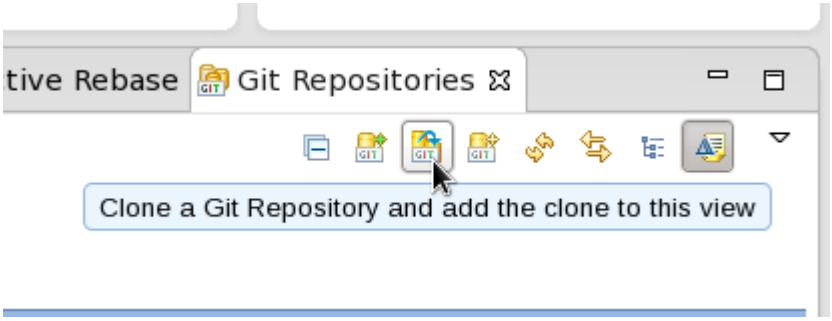
In most practical situations you will need to get access to an existing remote Git repository. This is usually done from command line using the clone command: "git clone <uri>" will create a local folder containing a Git managed file system including a .git folder.
In general it is a good idea to have one folder on the local file system containing all Git repositories.

An example structure is the following:

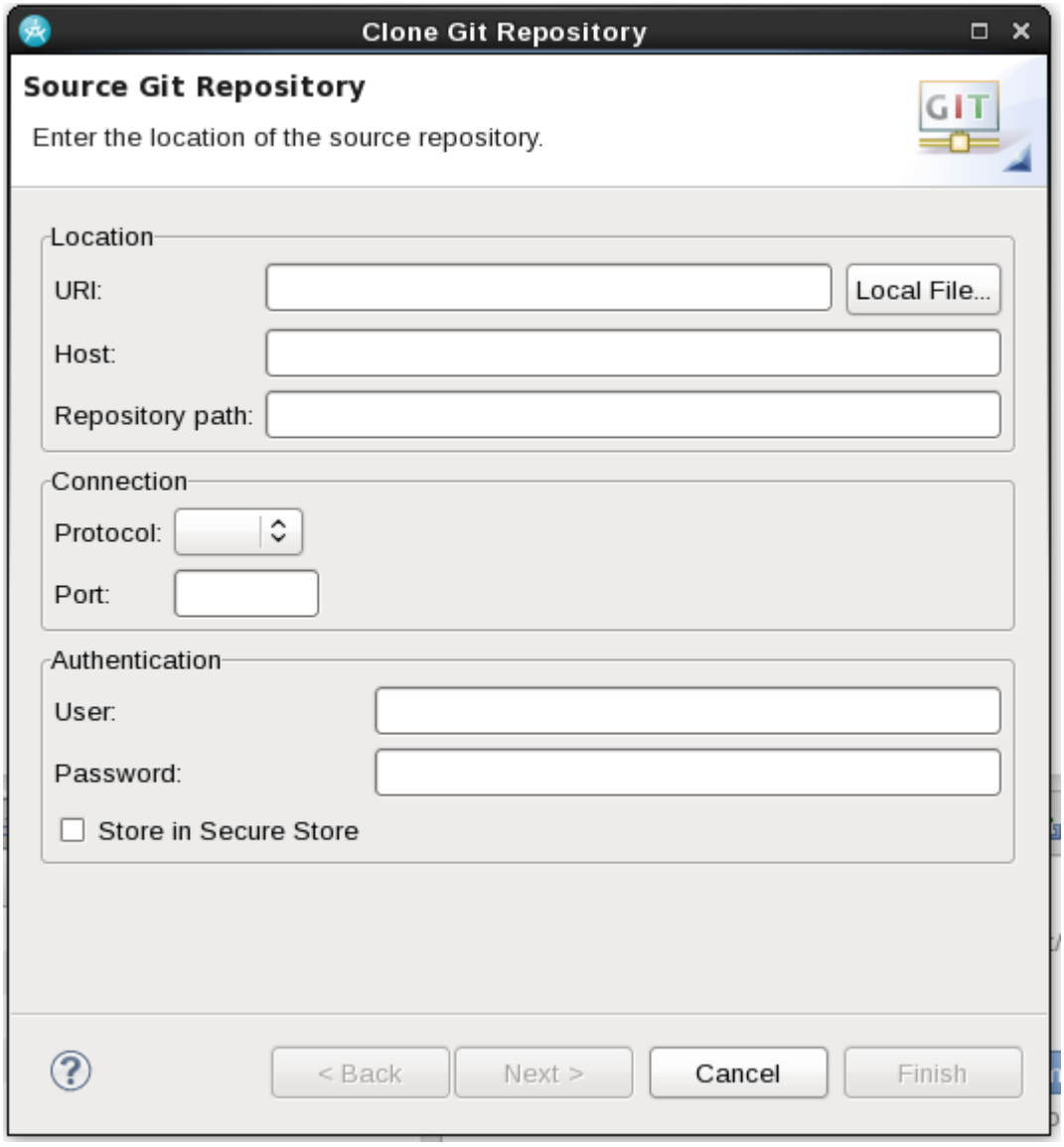
- /home/user/git
  - gitRepo1
  - gitRepo2

Whenever you need to access another remote Git repository just start a command shell in the directory corresponding to /home/user/git and do a "git clone <url>".

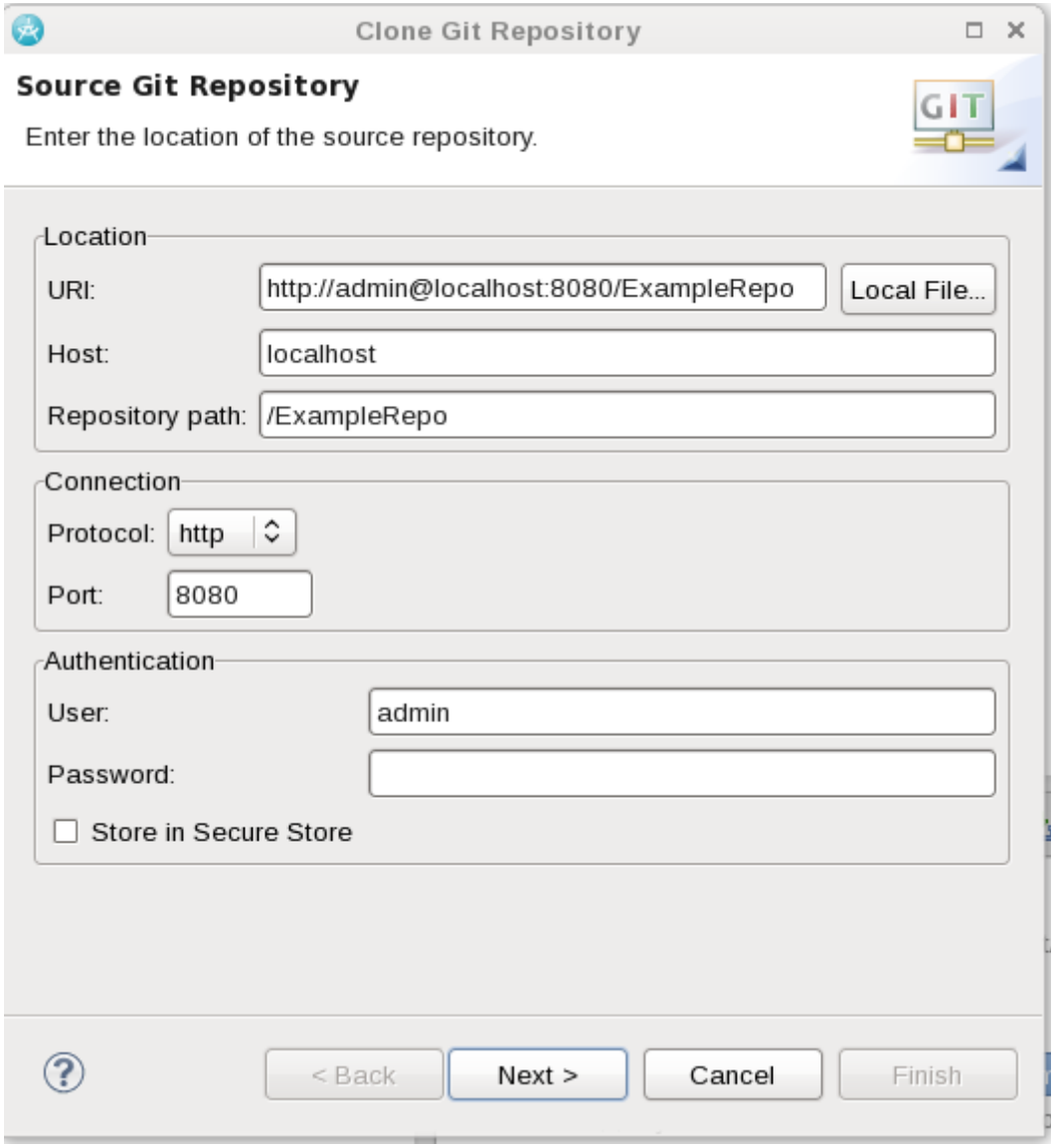
It is also possible (and convenient) to clone the repository from inside RSARTE. This can be done in the Git Repository view.



The dialog that appears makes it possible to specify the source of the repository.



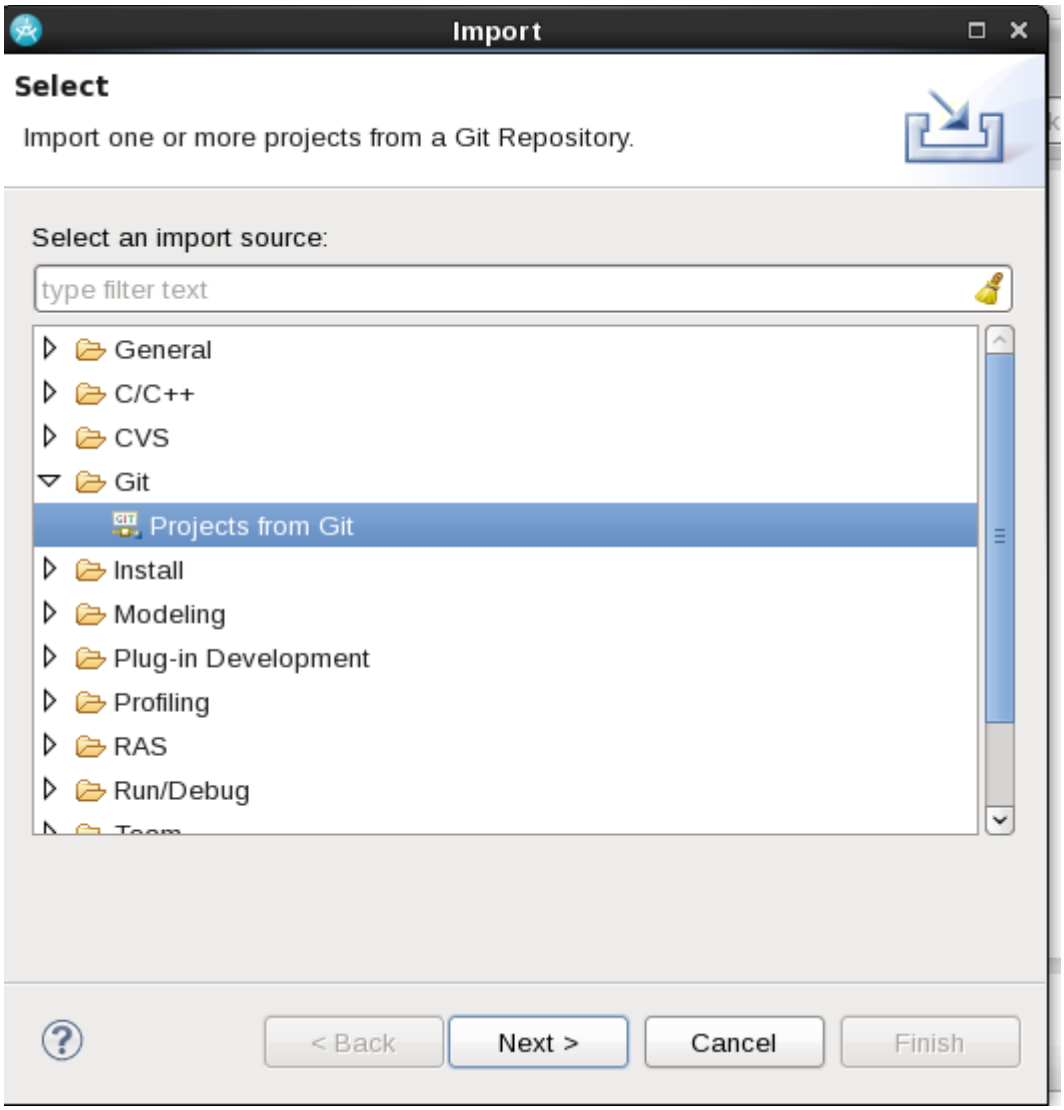
The easiest way to fill in the dialog is to simply paste a url into the top field. Then most of the other fields will be handled automatically. Adding ”http://admin@localhost:8080/ExampleRepo” to the dialog above will give the following result.

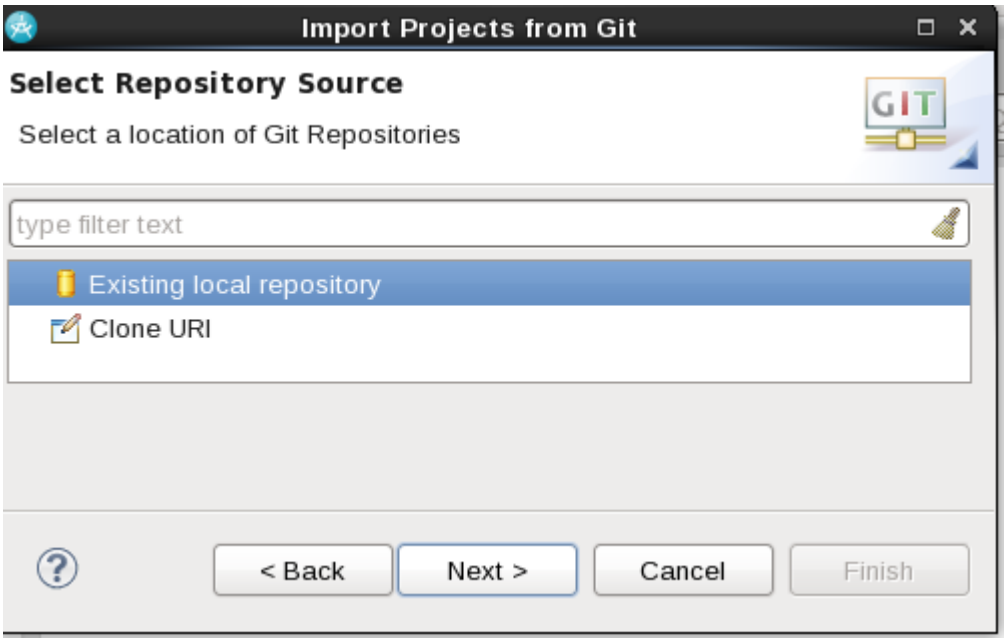


On the wizard pages that follow you can specify where in the local file system the repository will be located and also give a possibility to import any contained Eclipse projects into the current workspace.

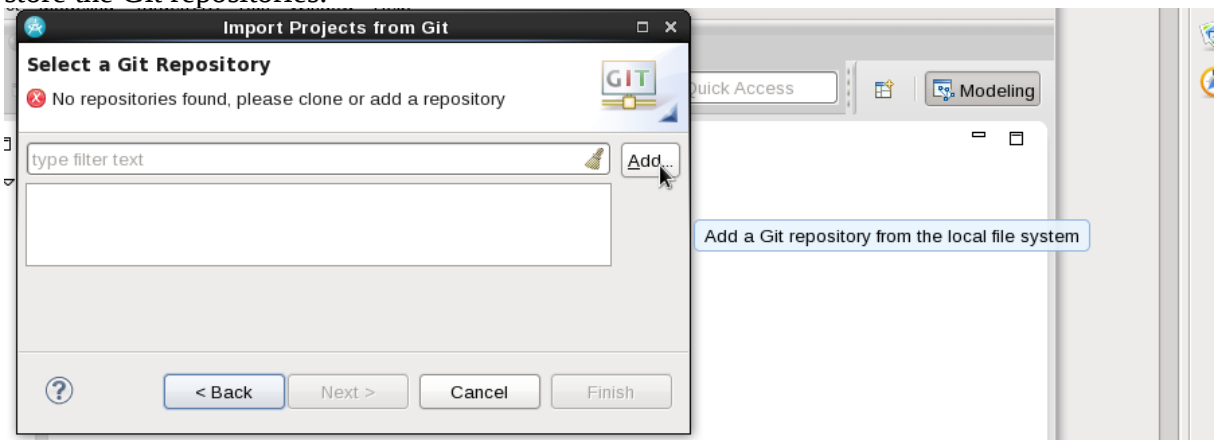
Importing an Existing Repository

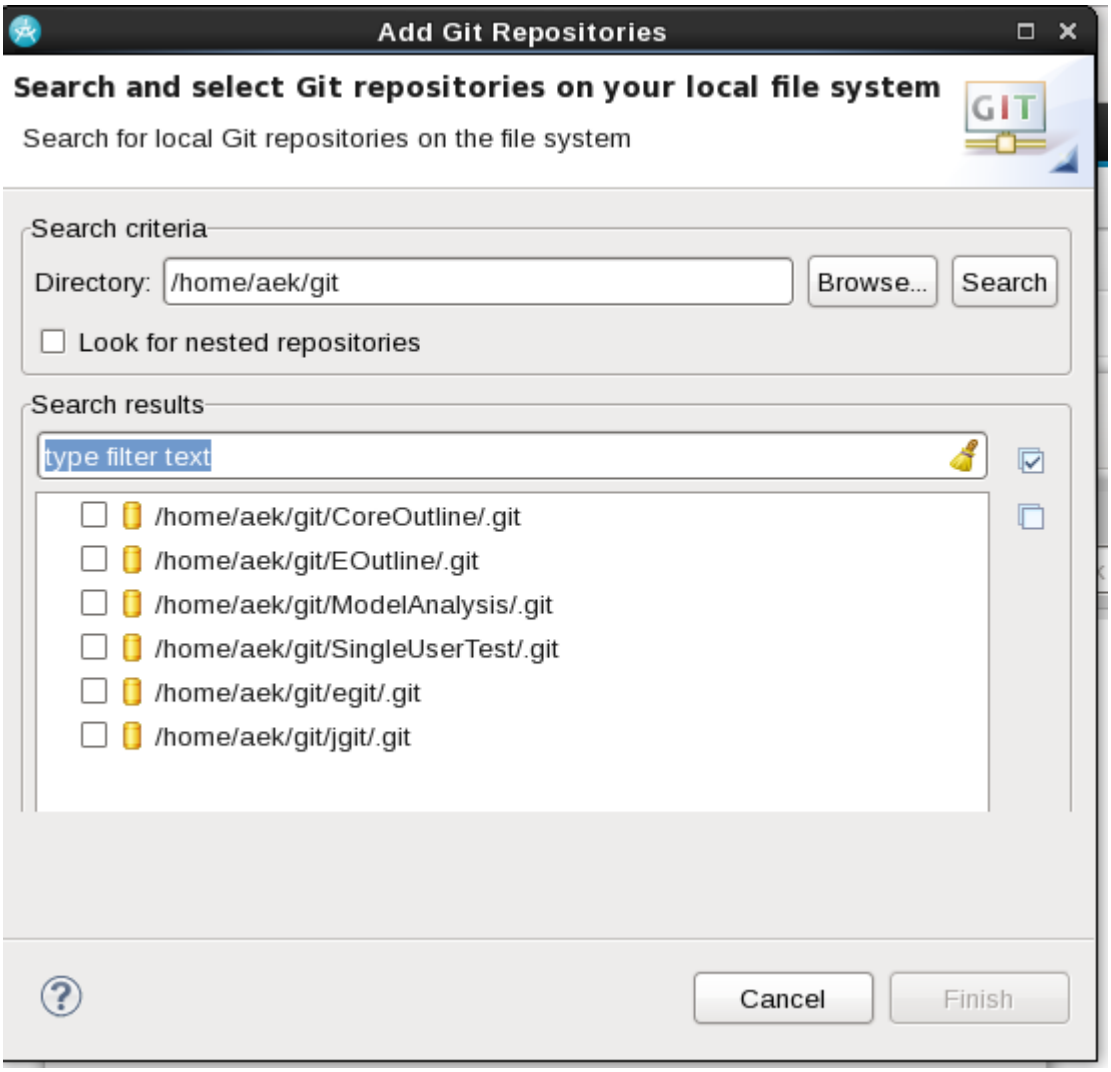
To get access to an existing local Git repository from RSARTE we can use the command File->Import... and choose ”Projects from Git”:


Import
Select
Import one or more projects from a Git Repository.
Select an import source:
type filter text
General
C/C++
CVS
Git
Projects from Git
Install
Modeling
Plug-in Development
Profiling
RAS
Run/Debug
< Back
Next >
Cancel
Finish

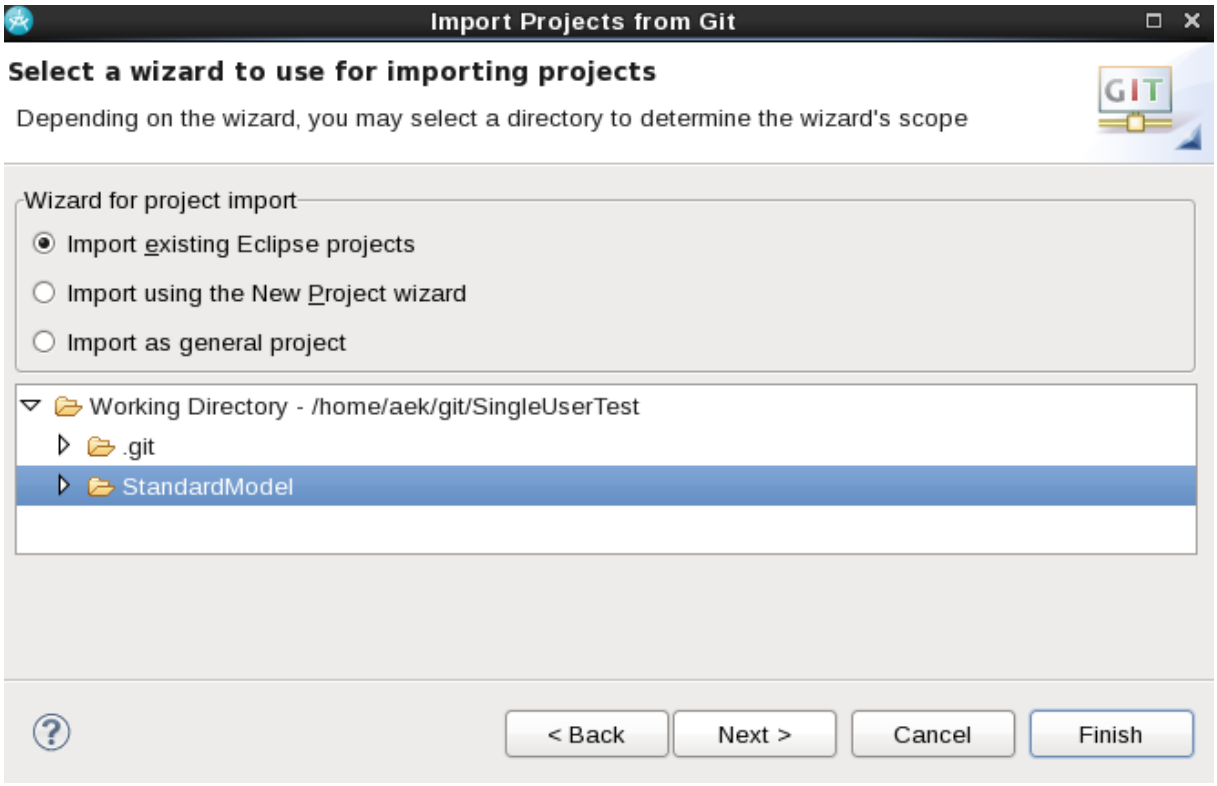

Import Projects from Git
Select Repository Source
Select a location of Git Repositories
GIT
type filter text
Existing local repository
Clone URI
< Back
Next >
Cancel
Finish

In the ”Select a Git Repository” dialog press ”Add” to inform RSARTE about where you store the Git repositories.





You can then select what projects to import and use the ”Import existing Eclipse projects” to perform the import.

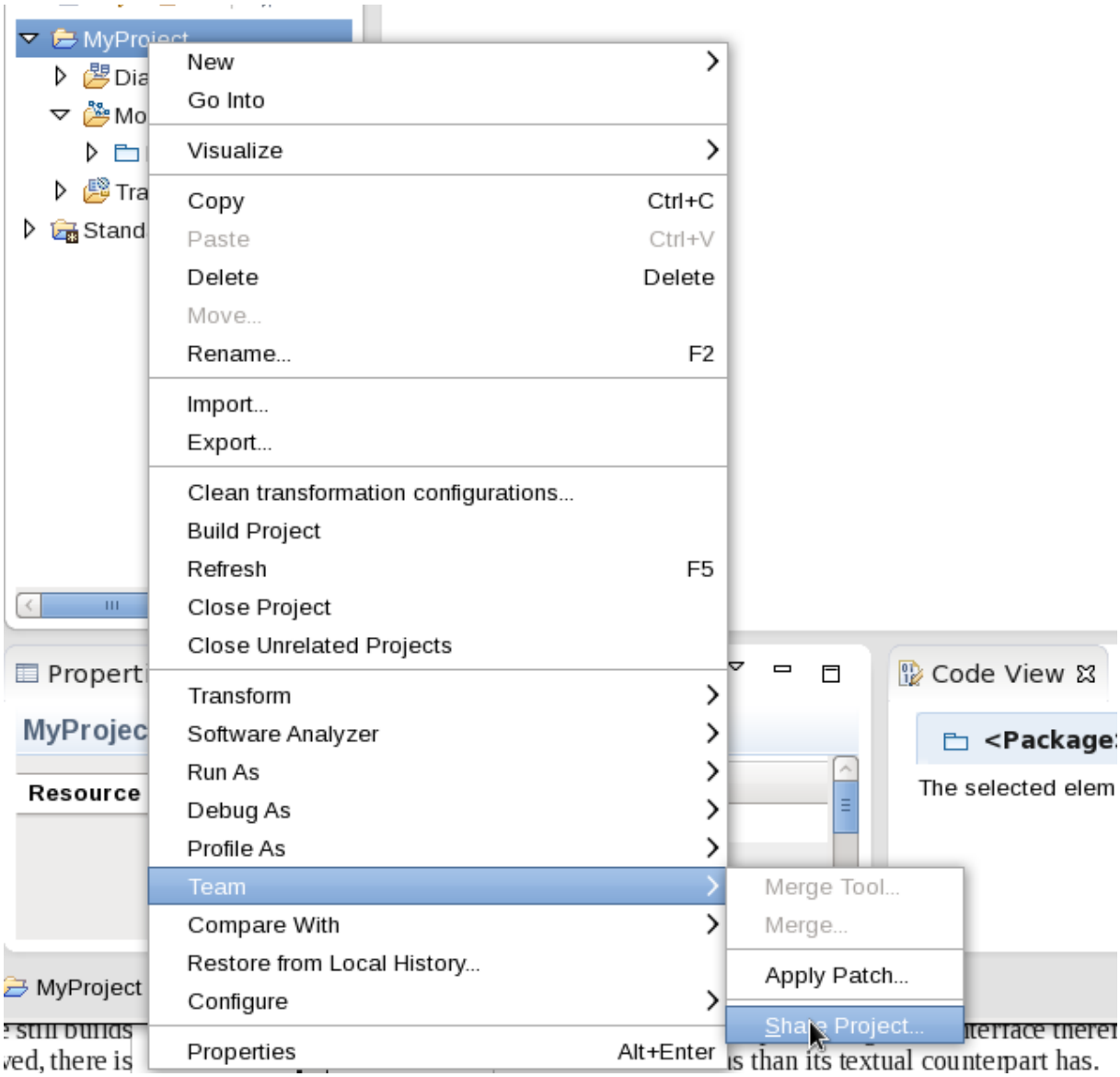

Import Projects from Git
Select a wizard to use for importing projects
Depending on the wizard, you may select a directory to determine the wizard's scope
GIT
Wizard for project import
Import existing Eclipse projects
Import using the New Project wizard
Import as general project
Working Directory - /home/aek/git/SingleUserTest
.git
StandardModel
< Back
Next >
Cancel
Finish

The result is a project available in RSARTE:

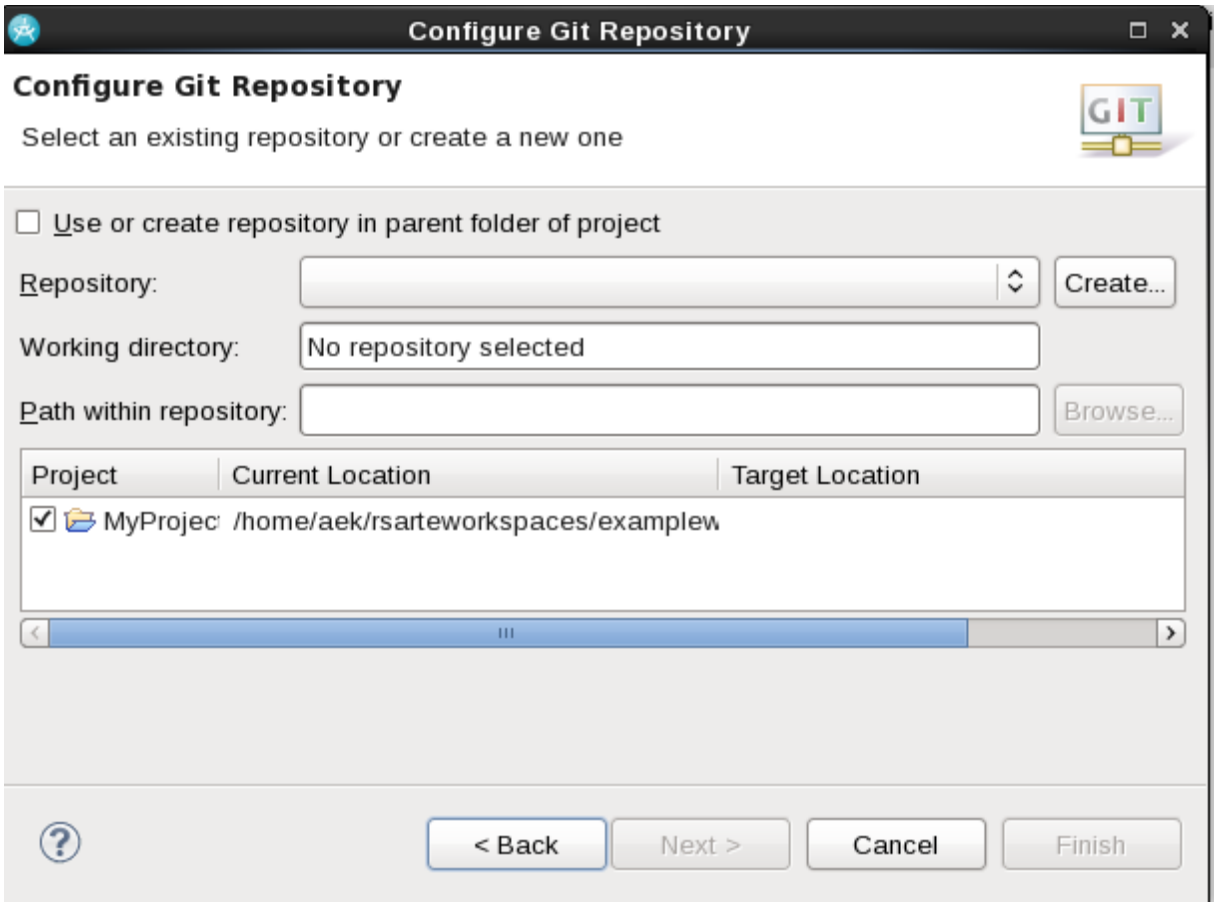
Note that this is a referenced project. The files are not stored in your Eclipse workspace, instead the files are stored in the Git repository. So, assuming that you have a directory ”/home/aek/git” that contains a ”SingleUserTest” Git repository containing a ”StandardModel” project as in the example above. Then if you create a new workspace and import the ”StandardModel” the files loaded in RSARTE will come from the directory ”/home/aek/git/SingleUserTest/StandardModel”.
This is important to remember since you most likely will use command line Git commands for various tasks and they need to be invoked in the Git repository in the file system.

Storing an Existing Project in Git

If you have an existing local RSARTE project and would like to use Git for version management this is of course possible. The easiest way to accomplish this is to use the ”Team->Share Project” available in the context menu for an RSARTE project.

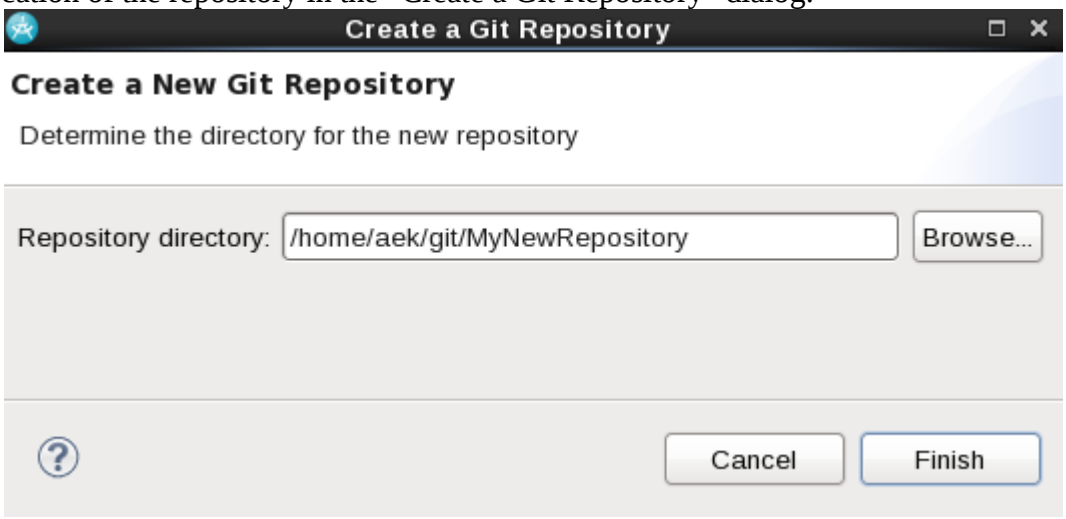


If you choose ”Git” as the repository type you can specify the details in the ”Configure Git Repository” dialog.

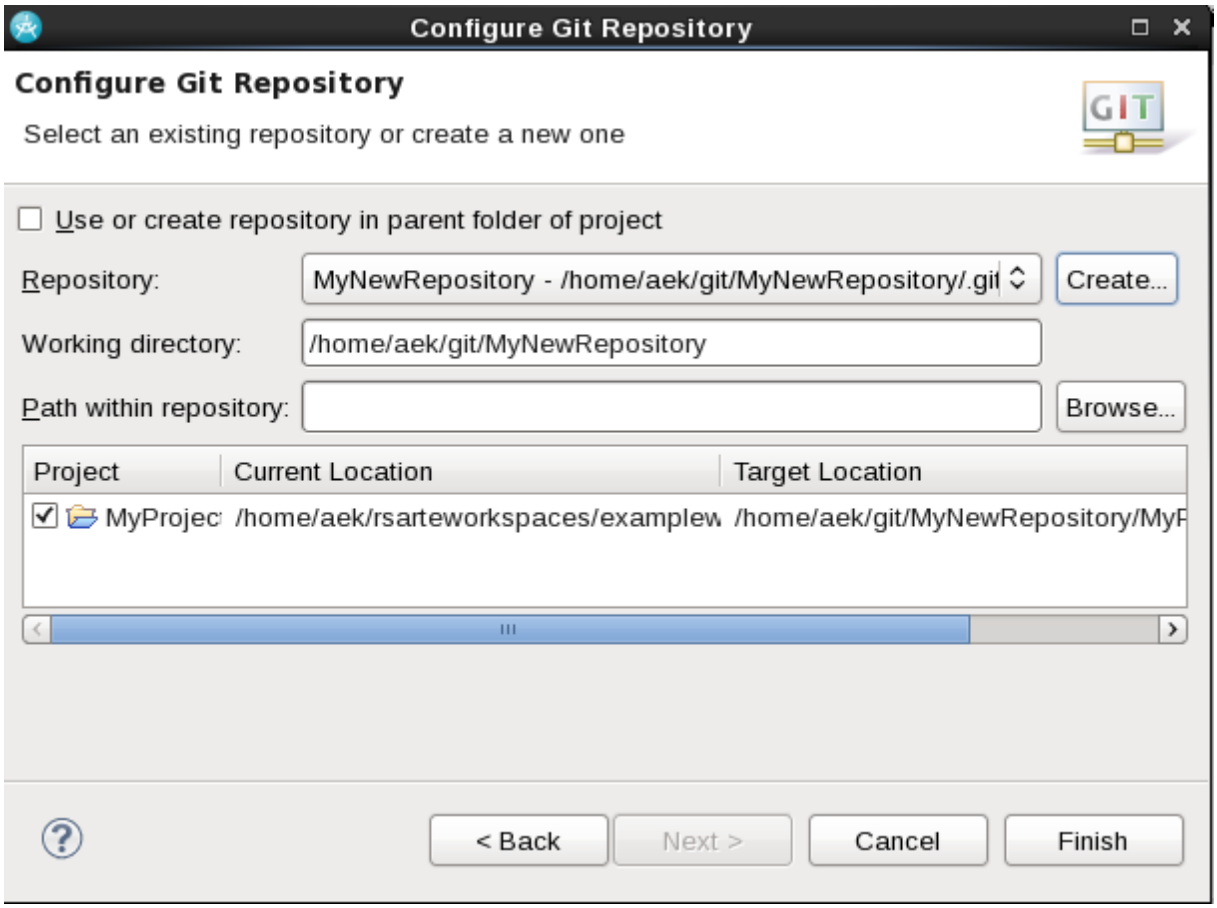


Since several different Eclipse projects can be stored in one Git repository the main task of this dialog is to select or create the Git repository.

If you press ”Create” you will get a new repository created and initialized. You can specify the location of the repository in the ”Create a Git Repository” dialog:



Back in the ”Configure Git Repository” dialog you can now see the details of the new repository that will be created.



If you now press ”Finish” several things will happen behind the scenes:

- A new Git repository will be created and initialized in the specified location
- Your project will in the file system be moved from your workspace location to the Git repository location
- Your project will be re-imported into RSARTE from the new location

You are now ready to start managing the project files using Git and EGit features. Note that one thing that does NOT happen automatically is to commit an initial version of the files to Git. See the discussion on the Git Staging View below for more information on how to accomplish this.

Git Staging – Checking work done and committing the result

When working with Git an operation that you frequently will do is to check the current status of your changed files. Typically you want to know:

- What in the model has been changed?
- What was the change?
- What is added to the Git index, ready to be committed.

For new Git users the ”index” concept in Git can be a bit confusing but essentially it is used to manage the Git representation of what will be committed to the Git history. From a practical point of view a file (or directory) in a Git managed file system can be in three modes:

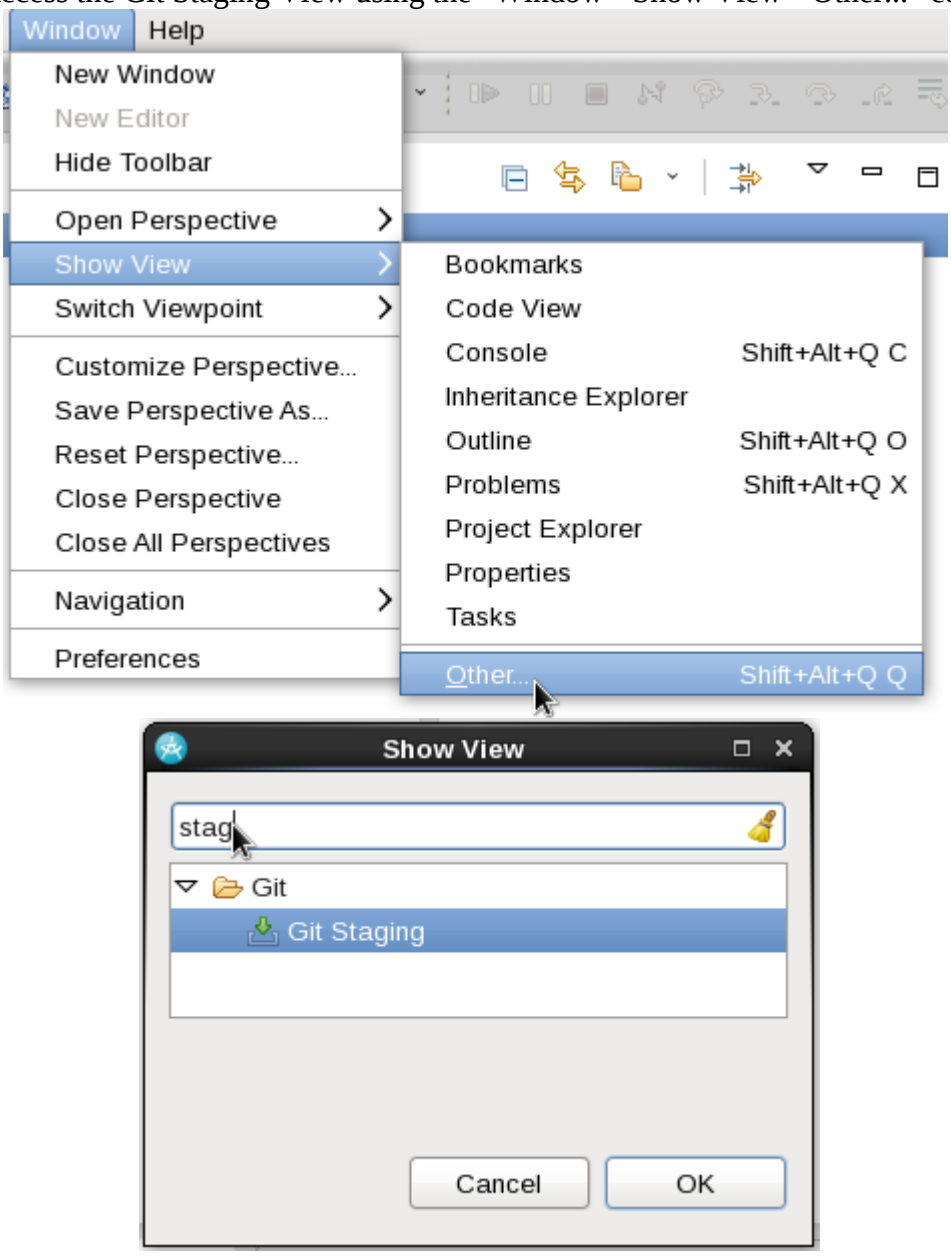
- Part of the local working directory. This is the version of the file that you can see and operate on using e.g. RSARTE.
- Part of the Git ”index”. This is nothing you typically will operate on, but is a temporary ”staging area” defining what will be committed to the Git history.
- Part of a ”commit”. This forms the version history of the files and are from most practical points of view read-only.

If using Git from command line the most commonly used command in this context is the ”git status” command. This will provide information about what files in your local working directory have been added or modified. It will also provide information about what changes you currently have added to the index.

When using RSARTE it is however in most cases more convenient to use the corresponding EGit functionality that is available in the tool. This is provided by the Git Staging View.

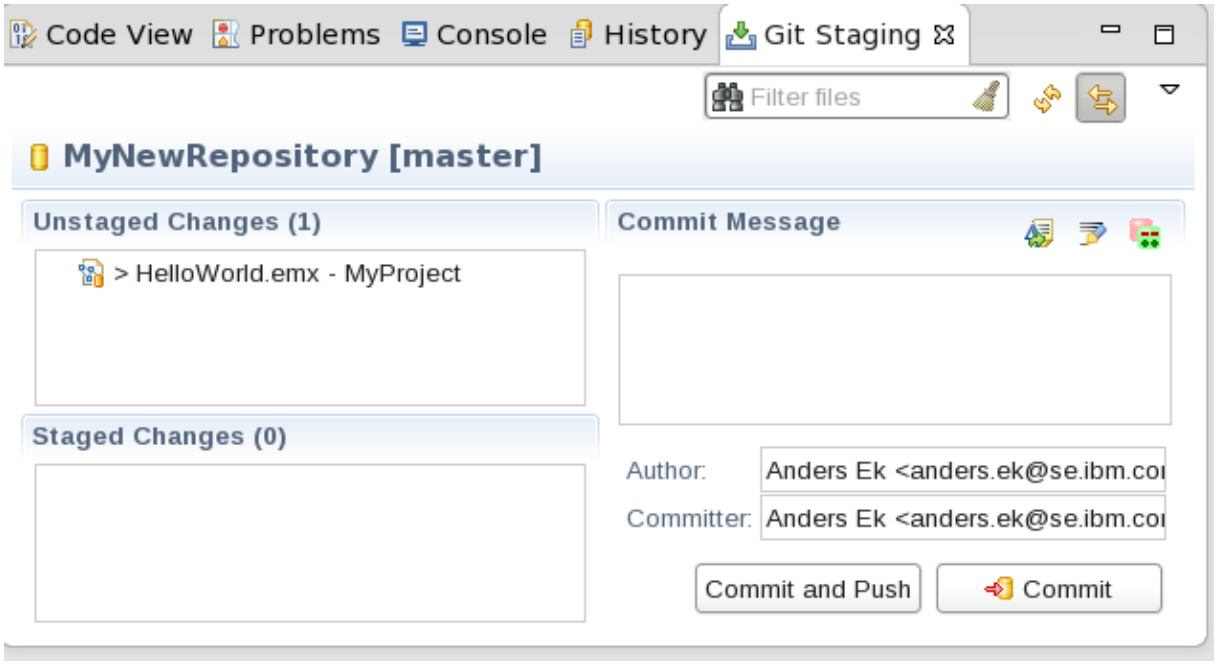
Git Staging View

You can access the Git Staging View using the ”Window->Show View->Other...” command.



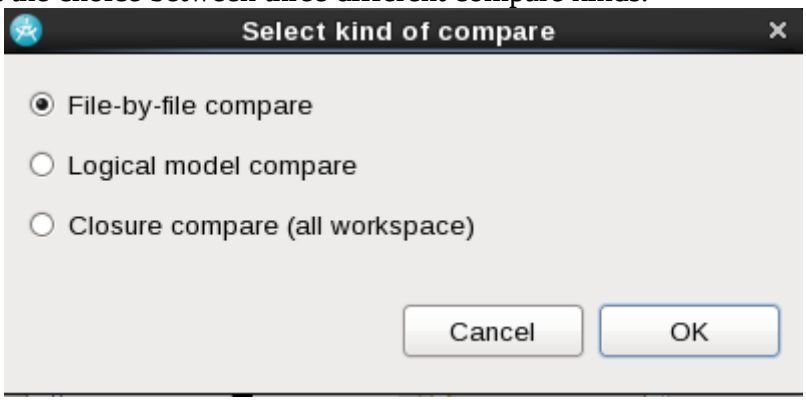
The Git Staging view has two main widgets:

- ”Unstaged Changes”: This is a list of files that have been changed in the local working directory
- ”Staged Changes”: This is the list of files that have been changed and added to the Git ”index”. These files will be committed to the Git history when pressing the ”Commit” button in the view.



Investigating Changes

To check the changes in a specific widget you can simply double click the file. If it is a model file you will get the choice between three different compare kinds.



- File-by-file compare will compare each file individually
- Logical model compare will compare a complete model, including the model file and all fragments together in one session

- Closure compare will trigger one compare session for all model files changed in the entire workspace.

See also the document Comparing and Merging Models for more information about different kinds of compare.

If you have many files in Unstaged Changes a convenient way to check all changes in model files is to double-click on one of the model files and choose ”Closure compare”. This will show all differences in one compare view.

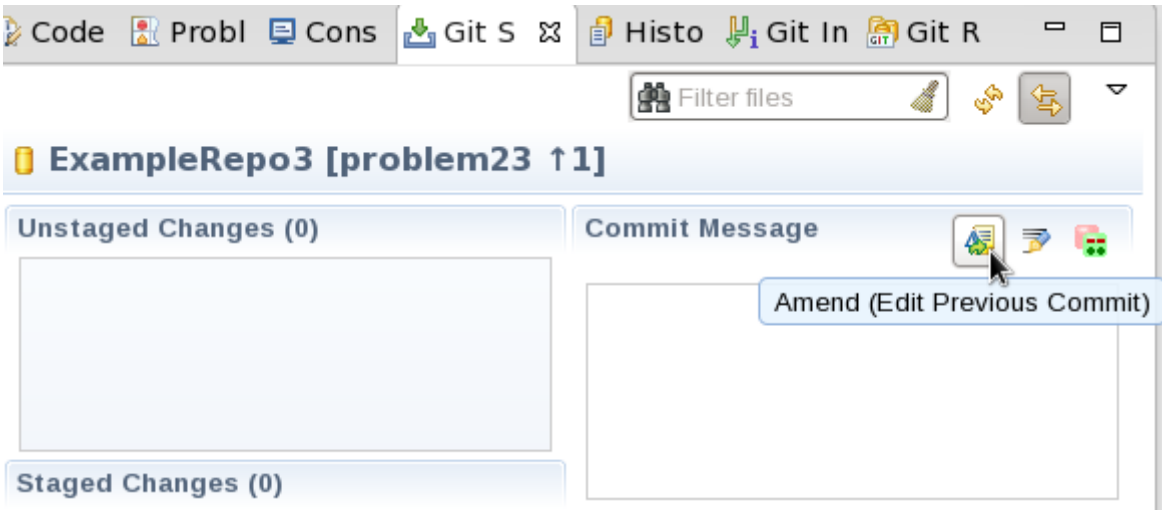
Committing Changes

To be able to commit your changes you first need to ”stage” them, i.e. add them to the Git index. This is done simply by dragging the files from the ”Unstaged Changes” list to the ”Staged Changes” list. If you change your mind you can easily ”unstage” them by dragging them in the other direction.

To commit the changes to Git history simply press the ”Commit” button, after describing the changes in a the ”Commit Message” area.

Amending Changes to Previous Commit

Usually when committing your changes you will create a new item in the version history of the repository. However, it is possible to instead modify the commit you currently have checked out. You can do this by clicking on the Amend button.



The visible result is that you get back the text in the Commit Message field for you to potentially edit. However more importantly you now have freedom to make any modifications

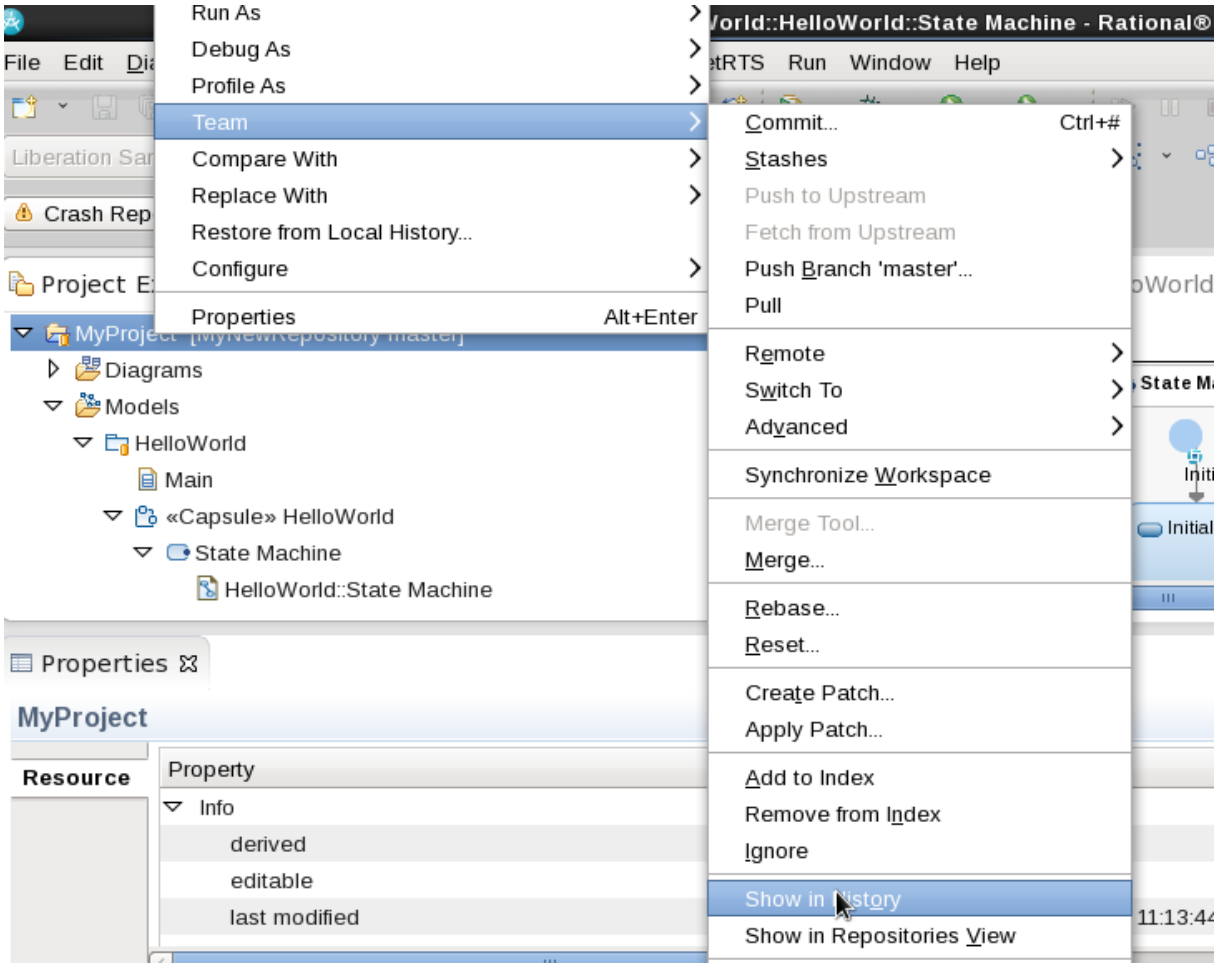
you would like including changing any files. When you commit, the result is not a new version in the history, instead you will replace the currently edited commit.

Git History – Investigating Commits and Branches

EGit provides RSARTE users with a convenient mechanism to investigate the version history of a Git repository: The Git variant of the History View.

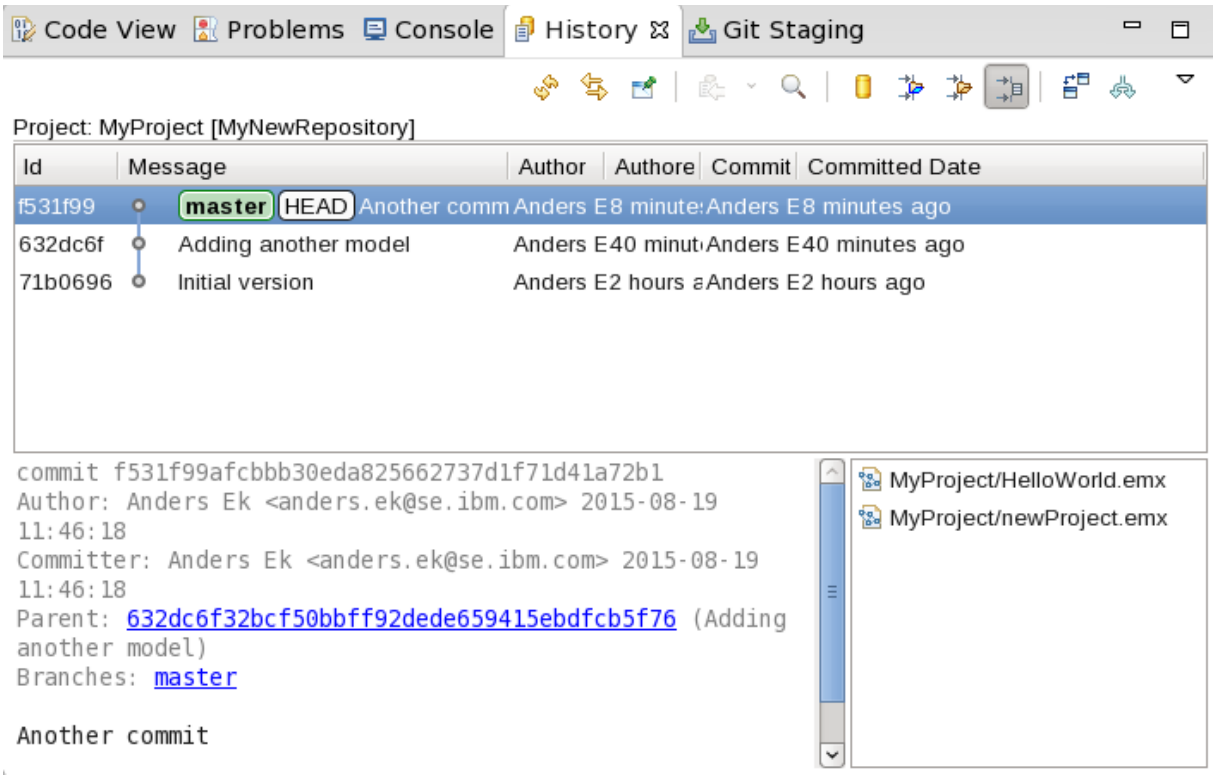
The History View

The simplest way to show the History view is to select a Git managed project in the Project Explorer and select the context menu command ”Team->Show in History”.



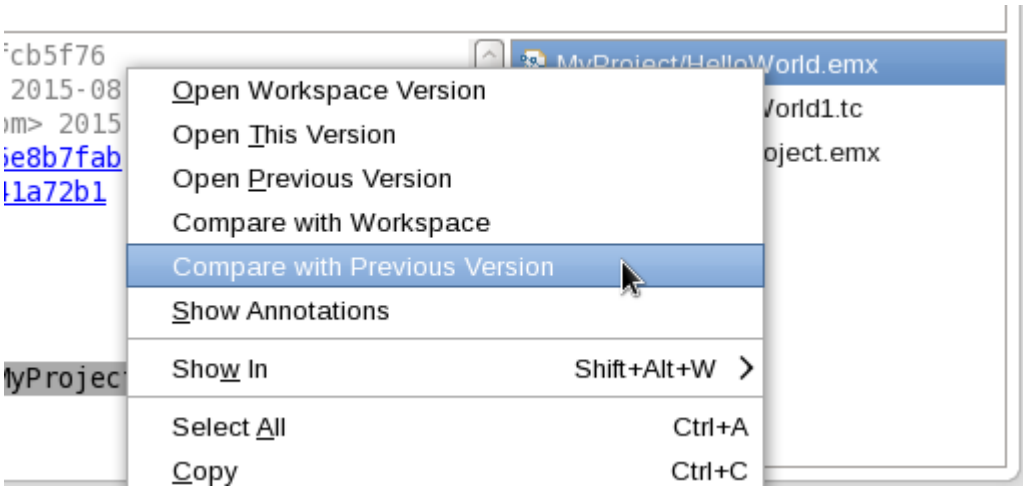
The History view has three main parts:

- The version history (top-most part)
- The file list (bottom right)
- The commit details (bottom left)



The basic workflow in the view is to select a specific commit in the version history and inspect the details using the file list and commit details parts.

To check the changes for a specific file and a specific commit use the compare commands available in the context menu.

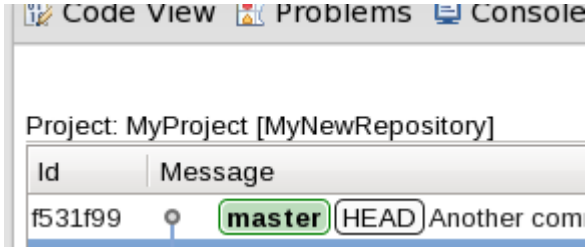


If comparing with the workspace both the file-by-file, logical and closure mode comparisons are available for model files. If comparing with previous version only file-by-file is available,

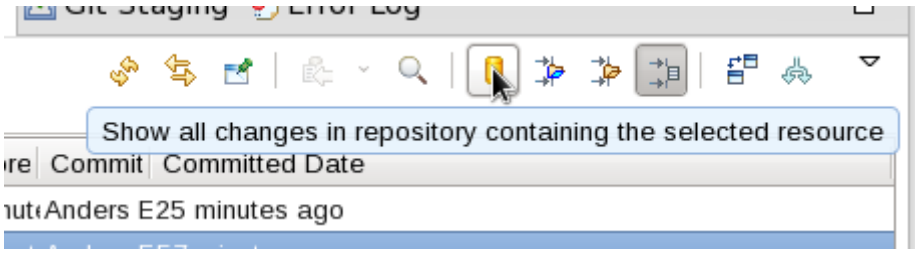
Filtering the History View

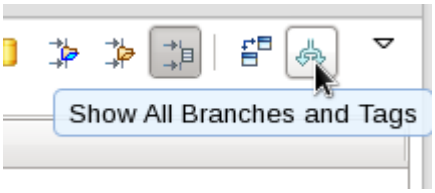
Several of the buttons in the History view are worth noticing, since you most likely will need to use them in your daily work.

If the version history does not show what you expect to see the reason is usually that the filtering is too narrow. The basics of the filtering in the History view is the focus element. What you see in the view is always filtered with respect to the element shown in the top left part of the view.



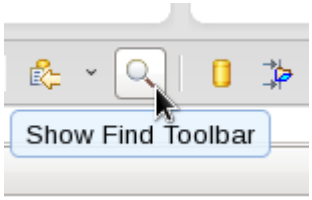
The filtering is controlled by buttons both with respect to repository elements and branches. To show as much as possible select both the repository scope button and the show all branches button.





Searching the History View

Another feature of the History view worth noting is the search possibility. You can start this by clicking on the ”Show Find Toolbar” button.



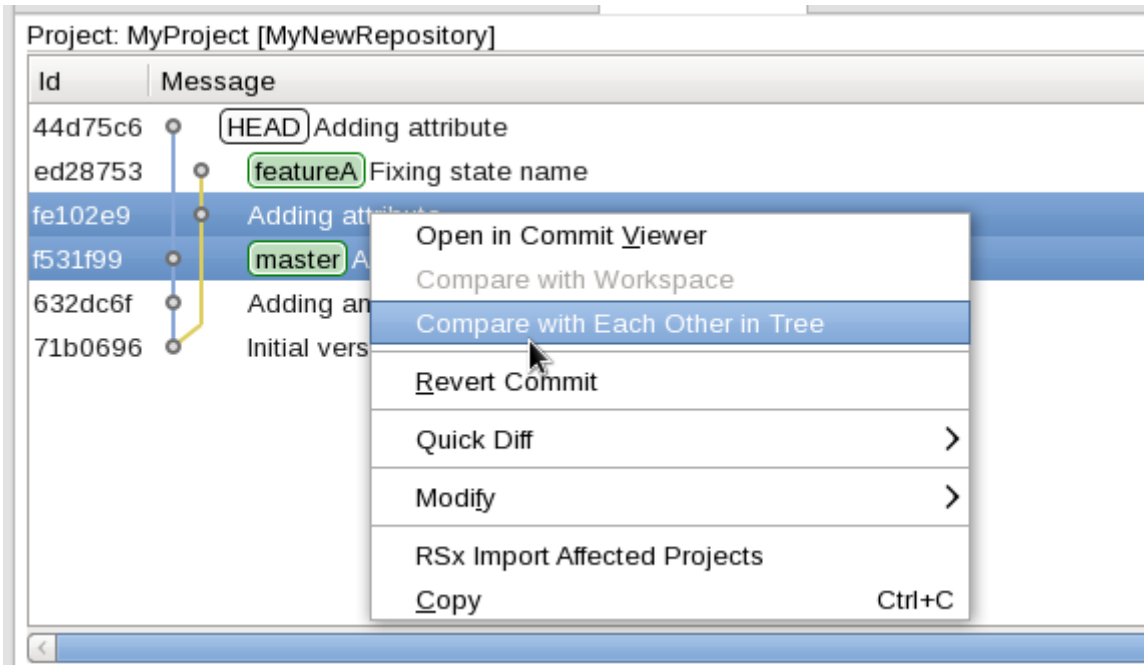
This will make the Find toolbar visible in the bottom of the view:

If you type something in the input box, a matching commit will be selected in the history and you can use the Next/Previous buttons to move to following or preceeding matching commits.

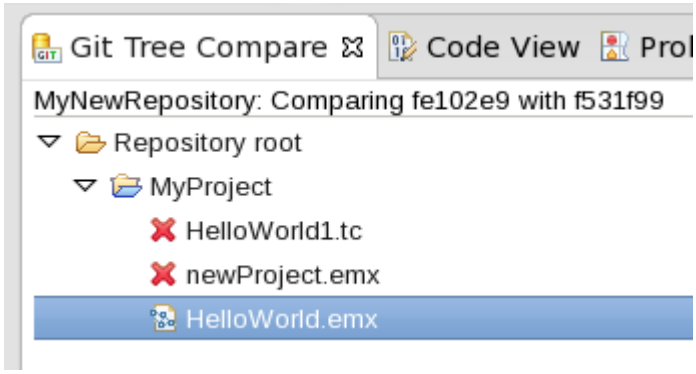
The pop-up menu also allows you to define what columns in the commit history are searched and if you want a case sensitive or case insensitive search.

Comparing Two Unrelated Commits – Git Tree Compare View

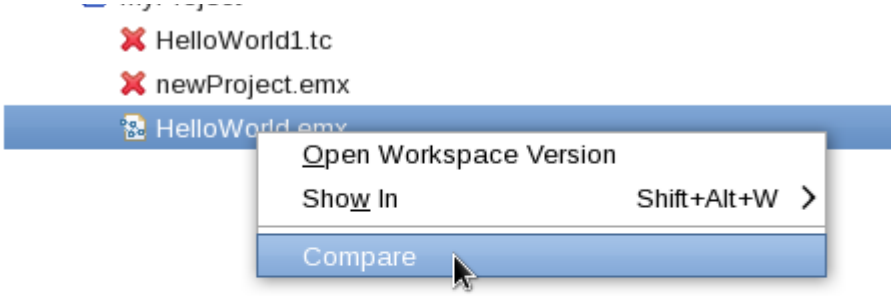
If you select more than one commit in the History view you will see an additional compare command in the context menu.



The ”Compare with Each Other in Tree” command will start the ”Git Tree Compare” view showing a file view of differences between the two selected commits.



From this tree view you can trigger compare sessions for the individual files.

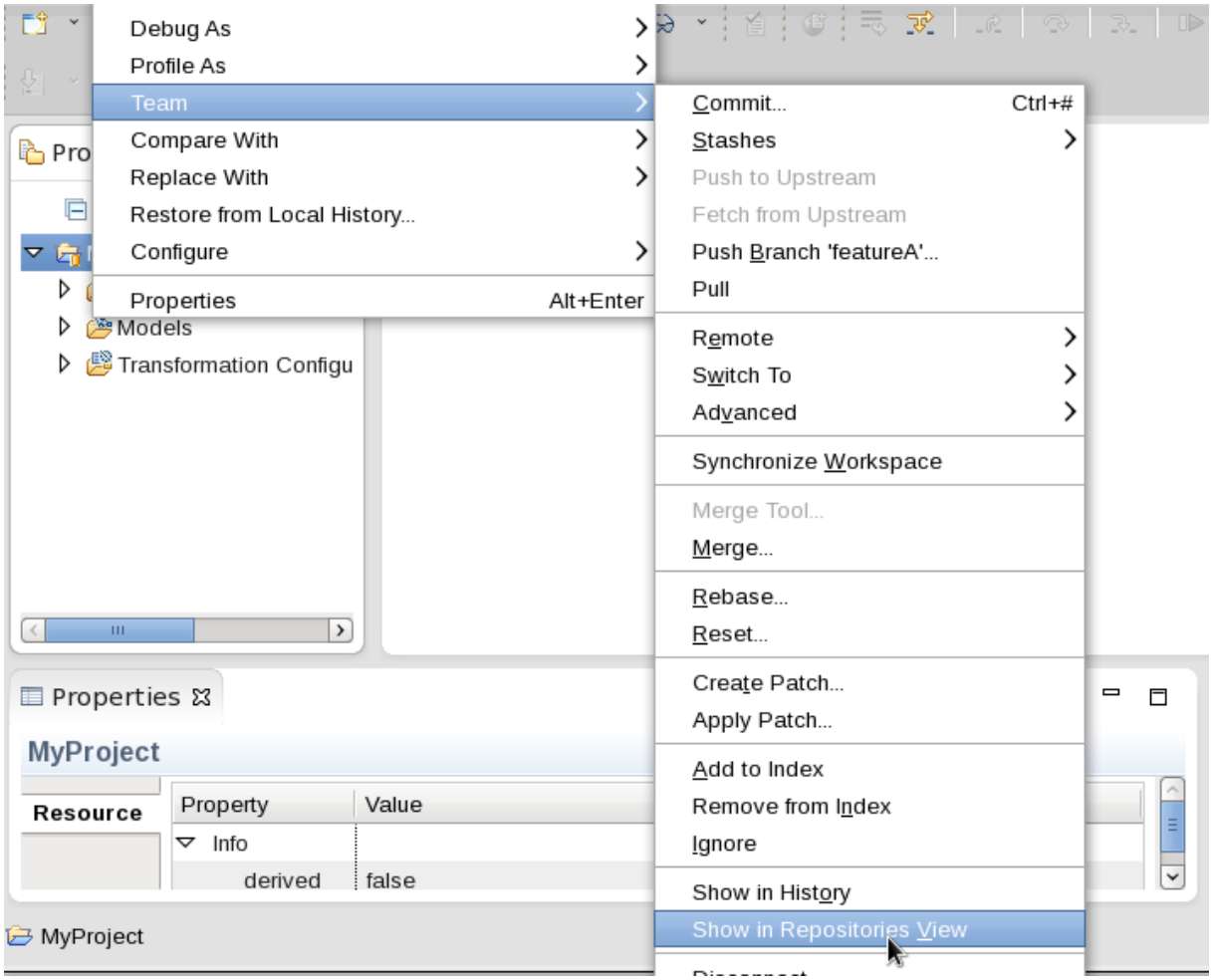


In most cases only file-by-file compare is possible, however if one of the versions you have selected is loaded in the current workspace also logical model and closure mode compare will be available.

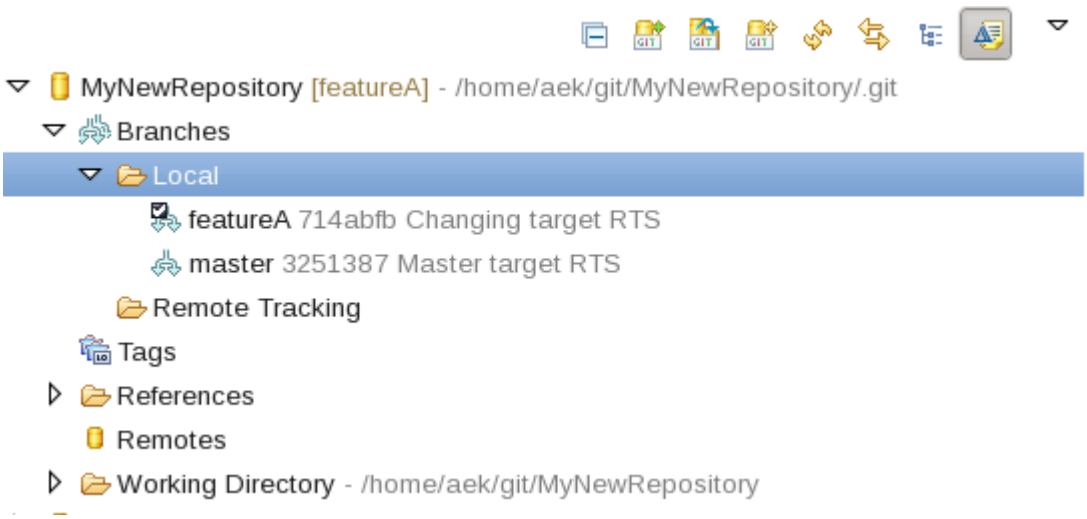
Using Git Repositories View to Investigate Branches

If you only are interested in an overview of the available branches and not the details of the history, then the Git Repositories view is a good alternative view. It provides a very concise description of what branches exist and which one you have checked out.

You can show the Git Repositories view from the Team command in the context menu on the Project Explorer when a model stored in Git is selected.



To investigate your local branches expand Branches->Local.



The branch with a small check-box is the branch you have currently checked out.

Checking Out and Creating Branches

To change the contents of your working directory to something from the Git history you need to check out a branch.

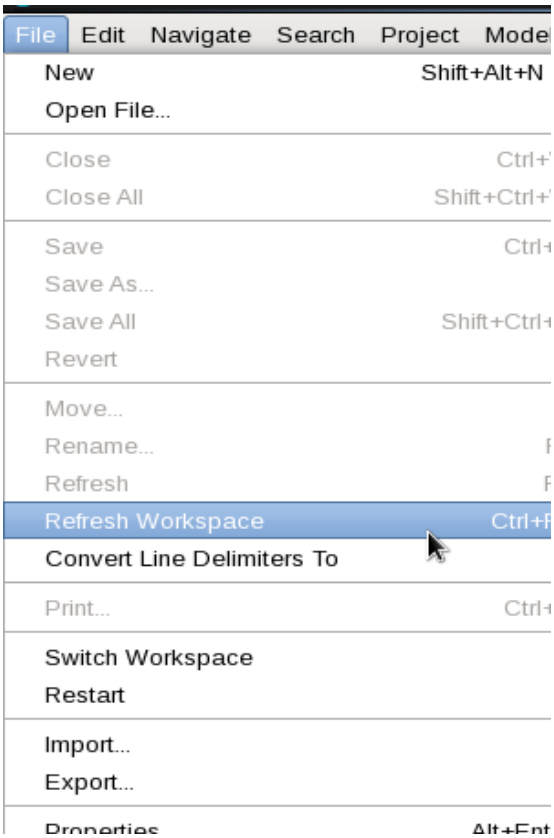
Check Out a Branch

It is possible to check out an existing branch both from command line and from the RSARTE user interface.

From command line the steps are typically the following:

- git branch
  - This command will list available branches
- git checkout <branch>
  - This command will replace the contents of the working directory with the version in the given branch. It will also tell Git to use the given branch as the starting point for whatever Git operations you will perform, like e.g. committing new or updated files

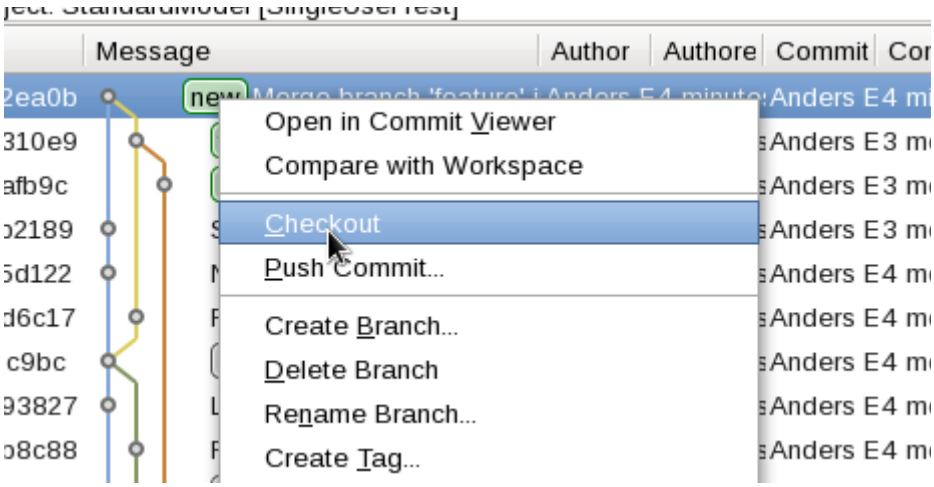
Note! If you check out a branch from the command line you need to make sure that you refresh the workspace before working with the loaded models to avoid problems. You can refresh the workspace using the ”File->Refresh Workspace” command. This will reload any model files you have opened and make sure that you have the version from the new branch loaded.



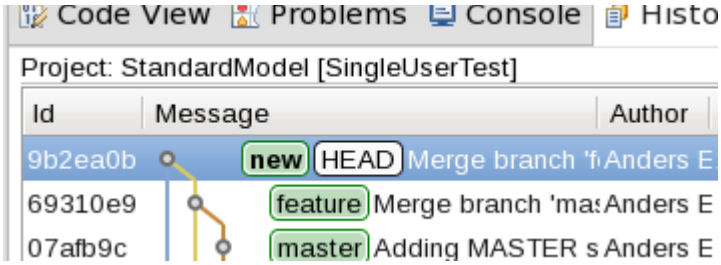
From the graphical user interface the most convenient way to investigate the branches is the History view. If you are looking for a specific commit you can use the Find Toolbar to search for it, otherwise just browse the history for what you are looking for.

The currently checked out branch is shown using bold characters and the current commit also has the indication ”HEAD”.

If you want to checkout a different branch you can use the command ”Checkout” available in the context menu in the history.



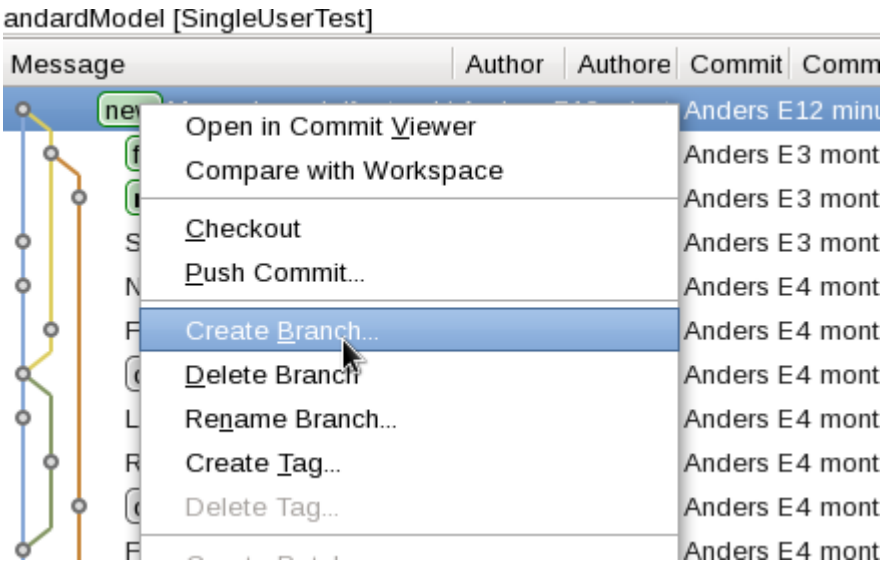
This will replace the contents of the working directory with the contents of the selected branch and also inform Git that this is the current commit, as indicated by ”HEAD”.



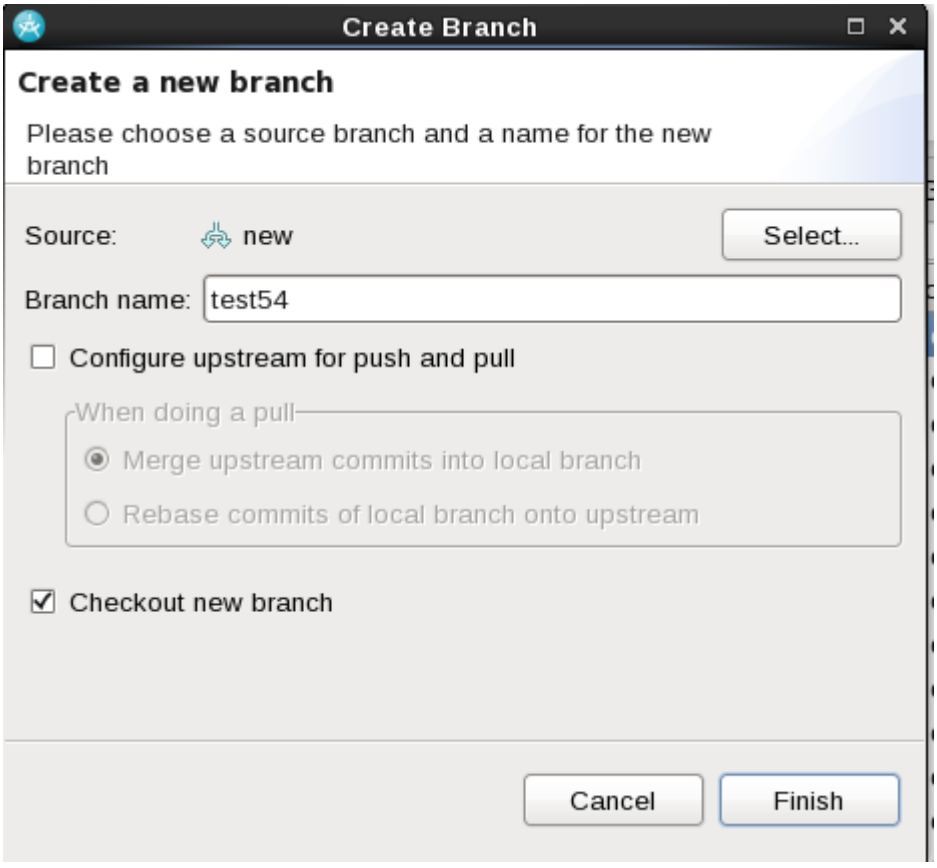
If using the user interface to check out a branch the workspace will automatically be refreshed and there is no need to use the ”File->Refresh Workspace” command.

Create a New Branch

To create a new branch from the History view simply use the command ”Create Branch” instead of ”Checkout”



This will prompt you for a name of the new branch and then create and check out the new branch.

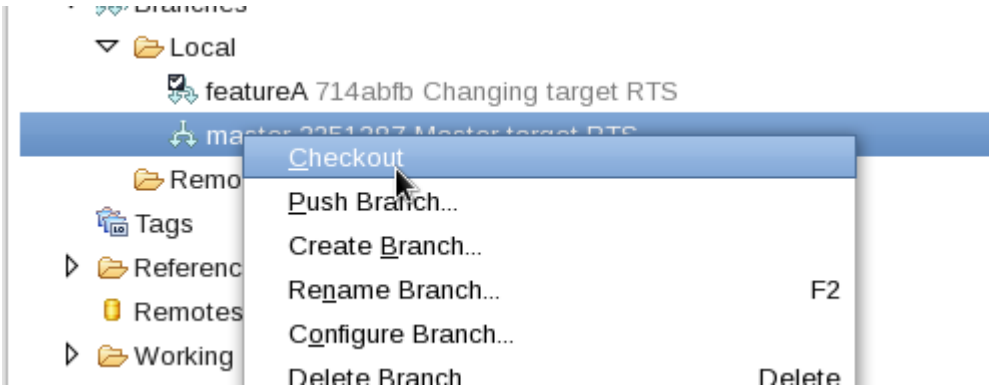


To accomplish the same from the command line first select the parent branch using ”git checkout <branch>” then create and checkout the new branch with ”git checkout -b <newbranch>”.

Using Git Repositories View to Check Out and Create Branches

If you already know what branch you want to check out and don't need to investigate the details of the different commits in the branch, then the Git Repositories view will allow you to quickly find and check out the branch you are looking for.

If you open the Git Repositories view as explained in Using Git Repositories View to Investigate Branches you can check out the branch you are interested in directly from the view.



You can also create a new branch from the same menu using the ”Create Branch...” command. The dialogs and actions are the same as when triggering this from the History view.

Merge and Rebase

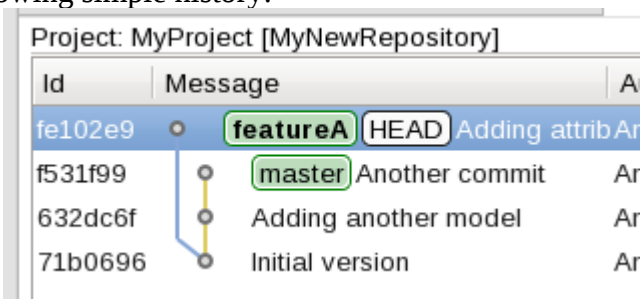
In most cases you will work on a specific feature branch when modifying the RSARTE model (or other files) and not on the ”master” branch to make sure that your changes during development are isolated from the rest of the world. Since branching as we saw in previous section is very easy using Git this is also the recommended practice. However, at some point you will need to catch up with changes done on another branch (usually the ”master” branch). This is done using the Merge and Rebase features of Git. Both of these commands will result in you having all changes from the ”master” branch available on your working branch. The main difference is in the deatils of how the resulting Git history will look.

- If using merge the changes on the ”master” branch will be applied as one additional change after the changes you have done so far on your branch
- If using rebase the changes you have done on your branch will be re-applied to the ”master” branch, after all other changes done on this branch

Both merge and rebase can be done both from the RSARTE user interface and from command line.

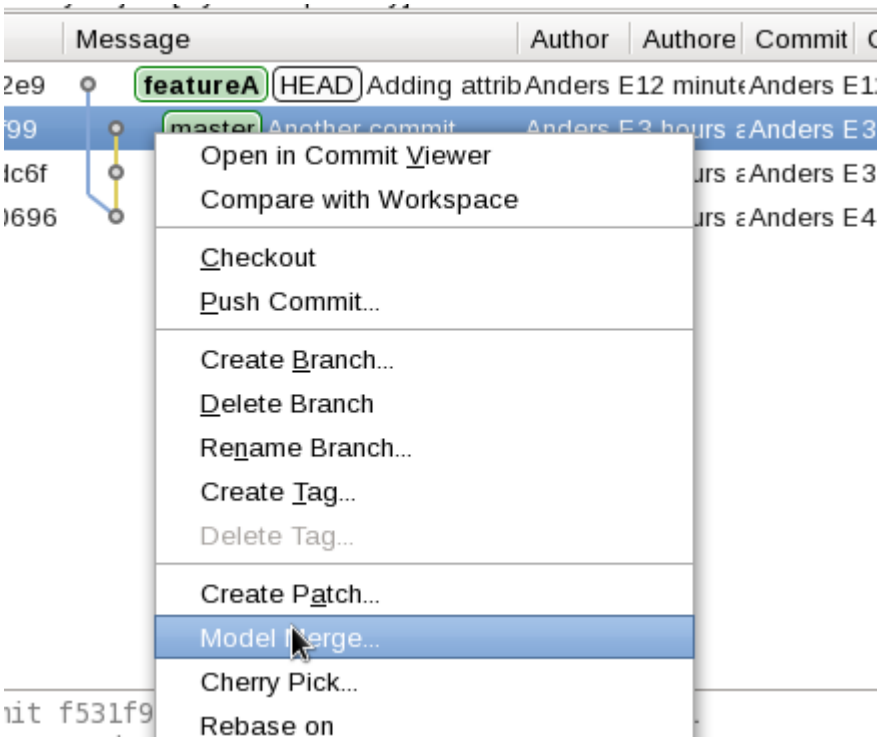
Merge

Let us start with a simple merge scenario. Assume that you are working on a feature branch featureA with the following simple history:

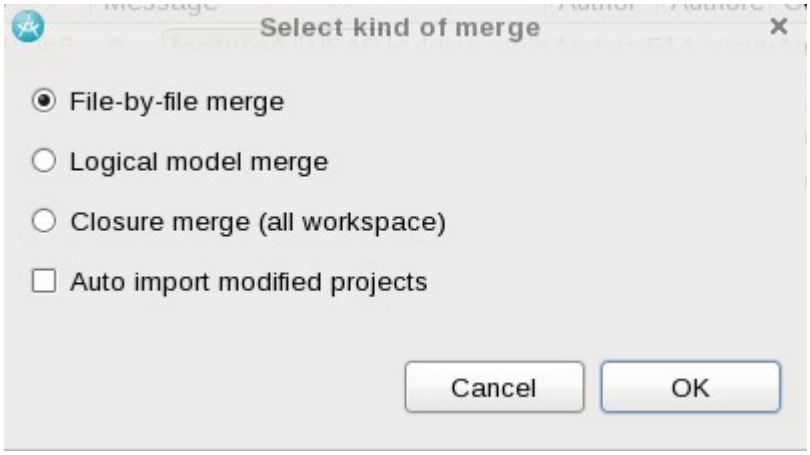


You now want to update your feature branch with the changes from ”master” using a merge.

Merge can be invoked from the History view using the command ”Model Merge...”. Note that even if the name implies merge of models the merge will also handle non-model files in the repository.



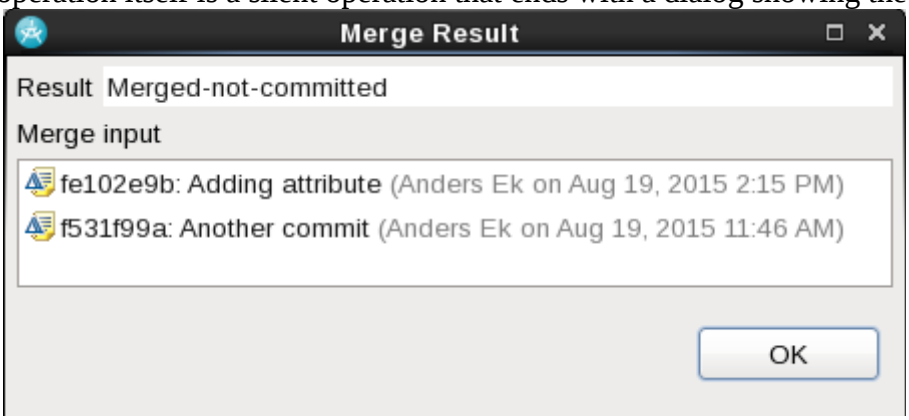
If using default settings you will get a dialog that allows you to select the merge mode used.



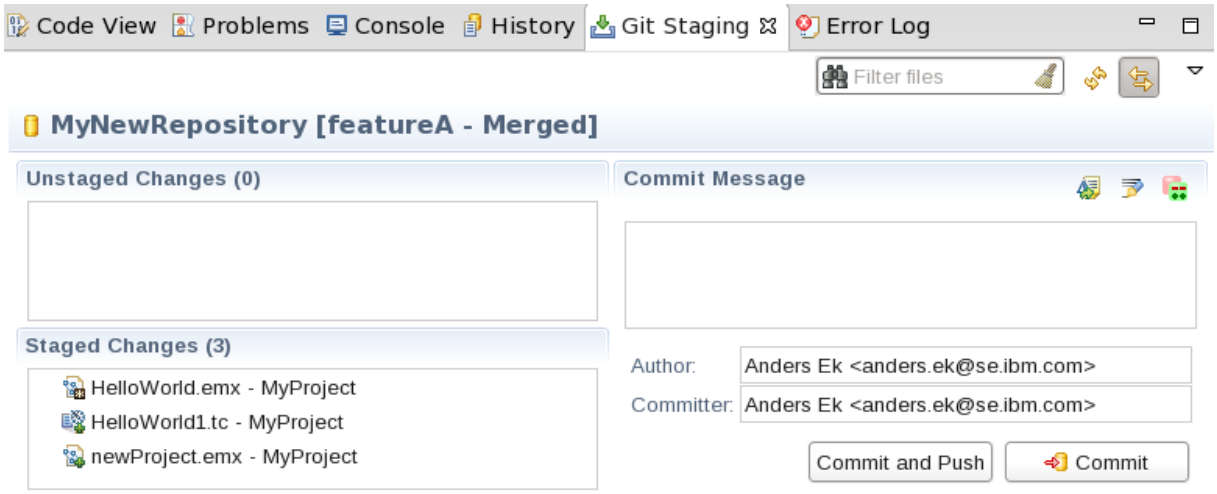
In most situations File-by-file merge is enough and it will also give best performance. However if you have done refactoring operations then Closure merge is recommended. See Comparing and Merging Models for a more detailed discussion about different merge kinds.

The ”Auto import modified projects” is a useful feature if you only have a small part of the complete model loaded in your workspace. If you select this option all projects that are modified will be automatically imported into your workspace. This will make it possible to inspect the changes that was done after the merge is completed.

The merge operation itself is a silent operation that ends with a dialog showing the result.

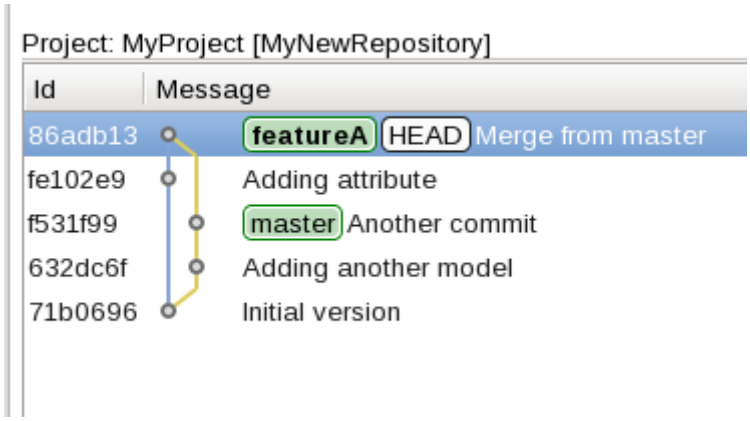

Merge Result
Result Merged-not-committed
Merge input
fe102e9b: Adding attribute (Anders Ek on Aug 19, 2015 2:15 PM)
f531f99a: Another commit (Anders Ek on Aug 19, 2015 11:46 AM)
OK

If there are no conflicts the new and changed files are added to the Git index and thus visible in the Git Staging view.



You can now check the changes caused by the merge as discussed in section Git Staging – Checking work done and committing the result and if they are as expected complete the merge by committing the changes.

If you then check the History view you can see that the merge from master branch is shown.



The same merge can also be done from command line.

A warning though: **Do not merge from command line if RSARTE has not been added to Git as a merge handler. Doing command line merges without adding RSARTE as a merge handler can often result in corrupt models.**

See section Integrating RSARTE with Command Line Git for more information about how to integrate RSARTE with Git command line merge.

The command line command for doing a merge from the master branch is simply:

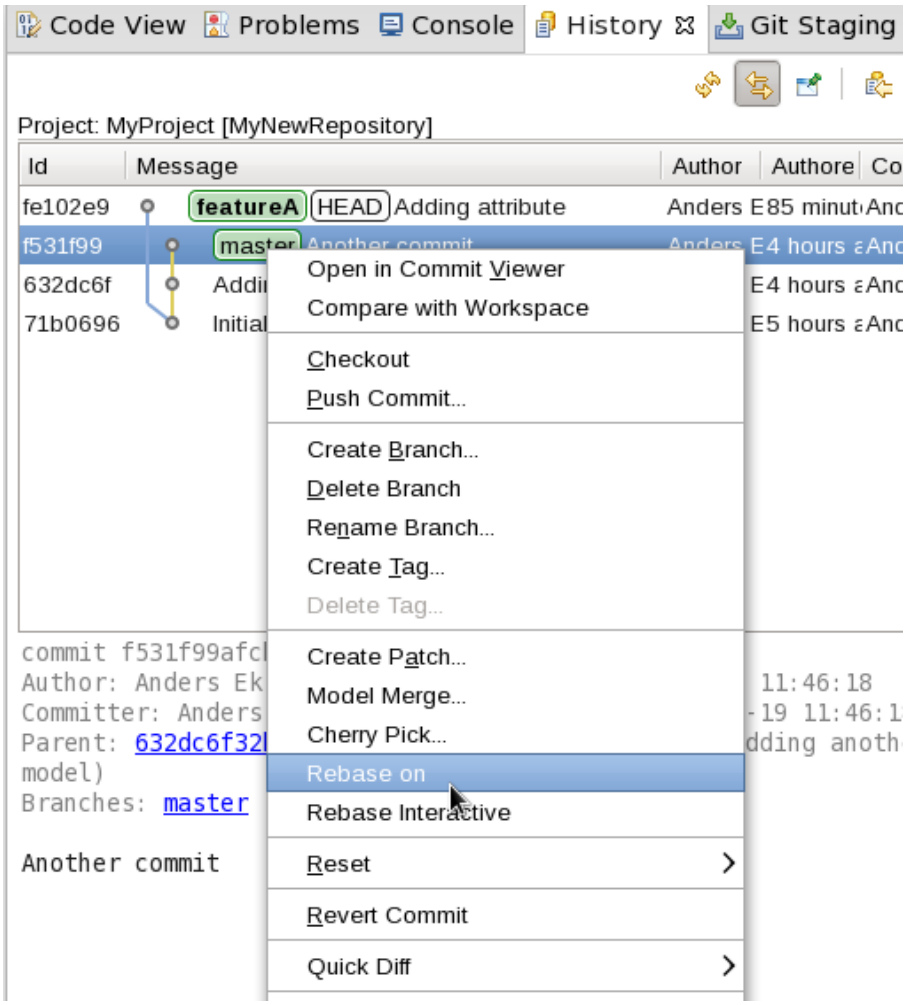
”git merge master”

This will in most cases trigger a file-by-file mode merge, but the details are controlled by the integration with the Git command line functionality as described in Integrating RSARTE with Command Line Git.

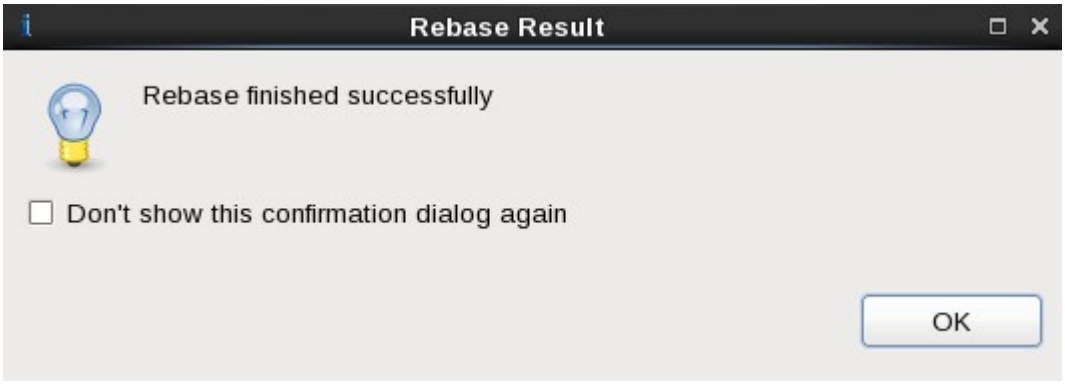
Once the merge is done you can use ”git status” and ”git commit” to check and commit the result. You can of course also mix the command line usage and RSARTE graphical user interface and do the merge from command line but use Git Staging View in RSARTE to check and commit the result.

Rebase

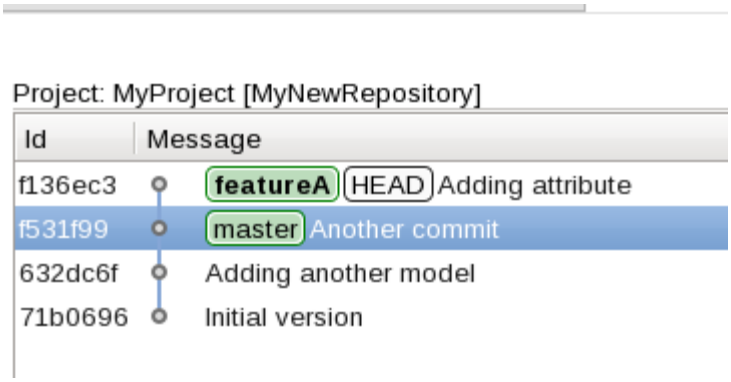
Doing a simple rebase from the GUI is similar to a merge. The easiest way to invoke it is to use the History view and invoke the ”Rebase on” command,



You will be prompted to choose the merge type and will then get a dialog with the rebase result.



The resulting history shows that instead of merging the changes from the master branch to the featureA branch, the changes you did on the featureA branch have been re-applied to the master branch.



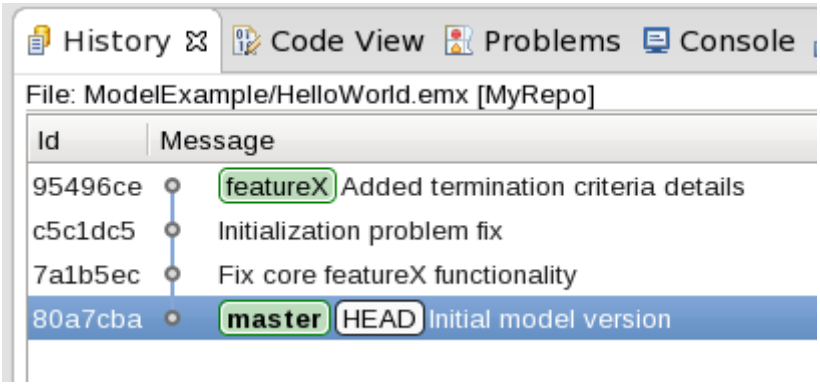
The rebase can also be invoked from the command line. The command to use is ”git rebase master” to invoke a rebase on the master branch. The same warning applies as for merge. **Do not invoke Git rebase for RSARTE model files without a proper integration into Git merge handling.**

Rebase is a possibility to create a nice and simple History view and is safe to use on local branches. **However, be very careful if using rebase on a branch that is shared with other team members (see Collaborating within a Team using Remote Branches). If someone else has downloaded the branch and have it in their local repository rebase will cause problems. So, if you work in a team, sharing branches, avoid rebase, or make sure to coordinate the rebase so everybody involved will clone a new copy of the repository after you have done a rebase.**

Rewrite History using Squash Merge

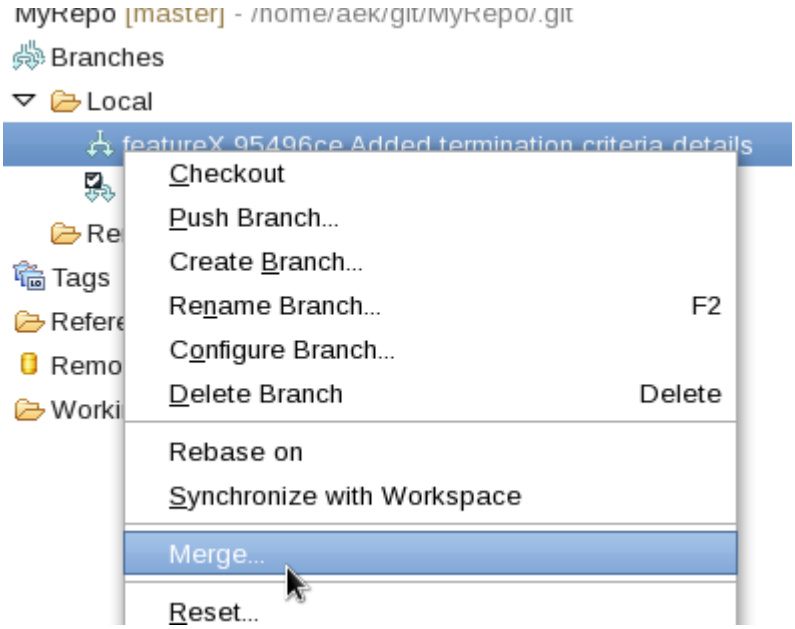
In a large organization it is important to have a reasonably simple version history with understandable commits. In this situation it may be the case that the granularity of commits on a feature branch is smaller than what we would like to have on the master branch. To overcome this problem Git includes a possibility to rewrite history during merge and collapse several commits into one commit. This is done using ”squash merge” and is typically done when merging changes from a feature branch to the master branch.

As an example assume that we have implemented a feature ”featureX” on a branch in three commits.

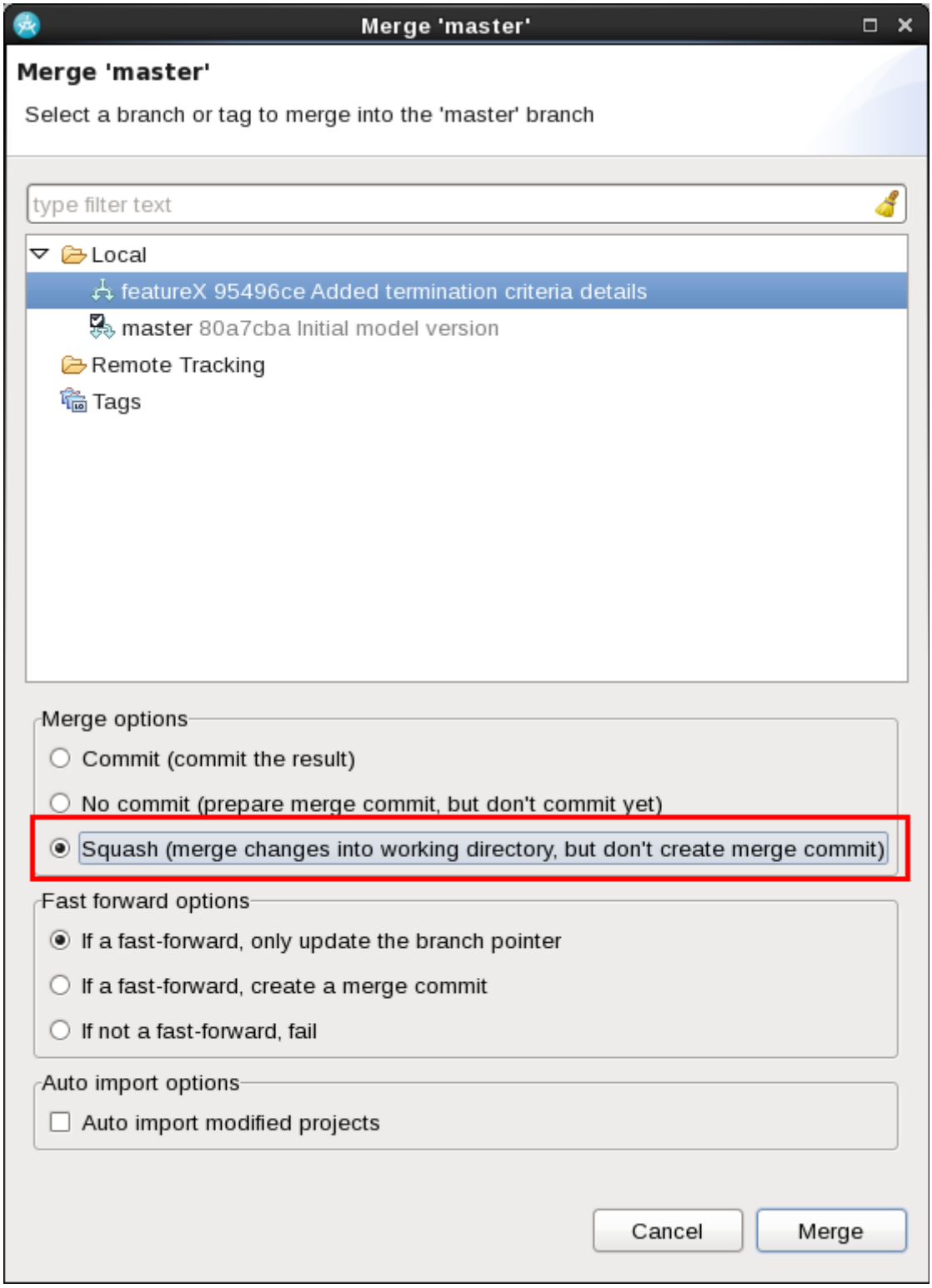


We now want to merge this into the master branch with a nice simple history.

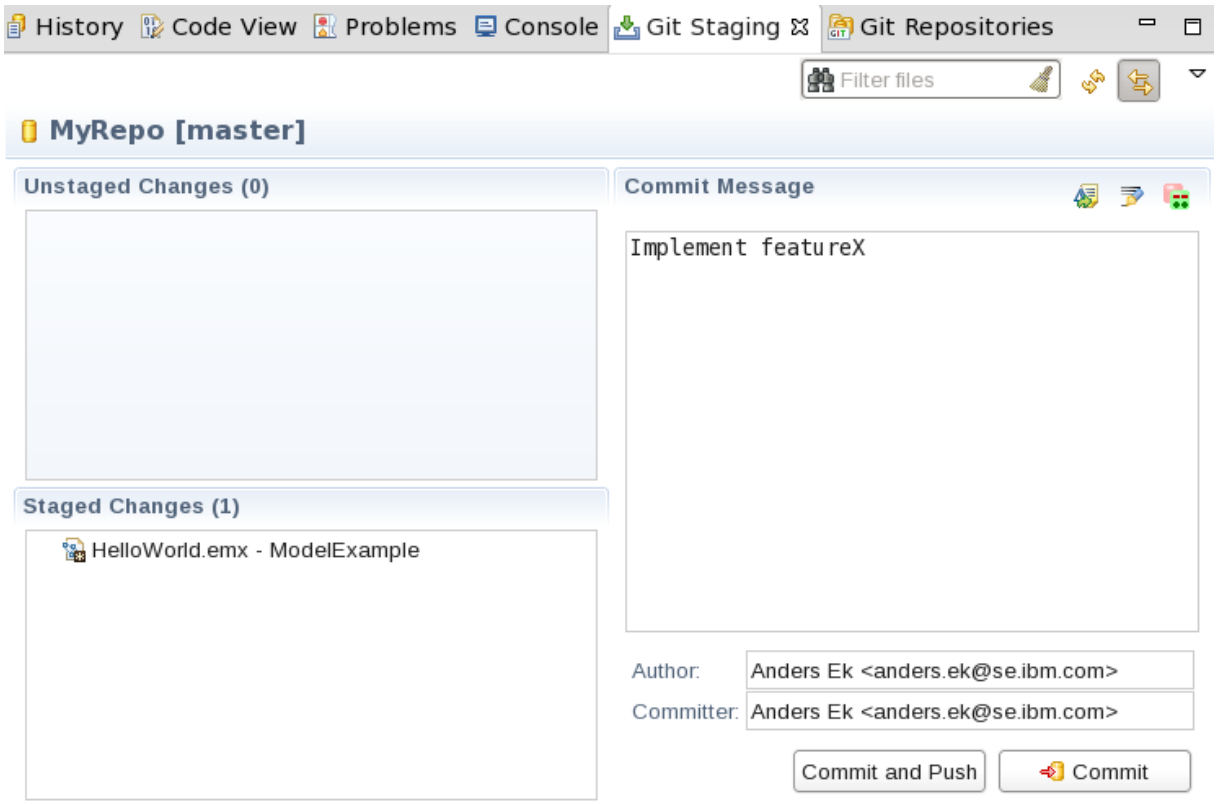
A simple way is to start by triggering the merge from the Git Repositories View .



Then select ”Squash merge” in the merge details dialog.


Merge 'master'
Merge 'master'
Select a branch or tag to merge into the 'master' branch
type filter text
Local
featureX 95496ce Added termination criteria details
master 80a7cba Initial model version
Remote Tracking
Tags
Merge options
Commit (commit the result)
No commit (prepare merge commit, but don't commit yet)
Squash (merge changes into working directory, but don't create merge commit)
Fast forward options
If a fast-forward, only update the branch pointer
If a fast-forward, create a merge commit
If not a fast-forward, fail
Auto import options
Auto import modified projects
Cancel
Merge

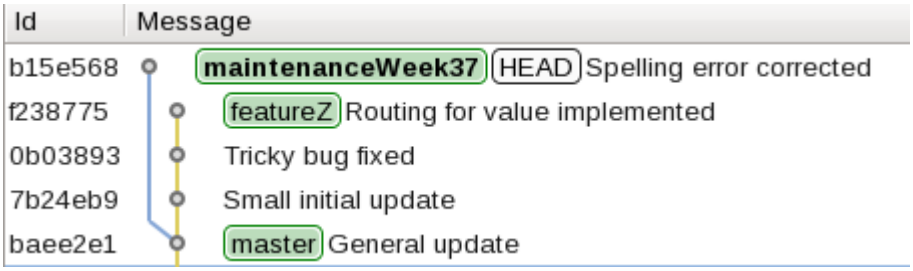
The result after the merge is that all changes from our feature branch are merged to master, but the result is not committed. This give us an oportunity to commit the changes ourselves, giving the change a suitable commit message.



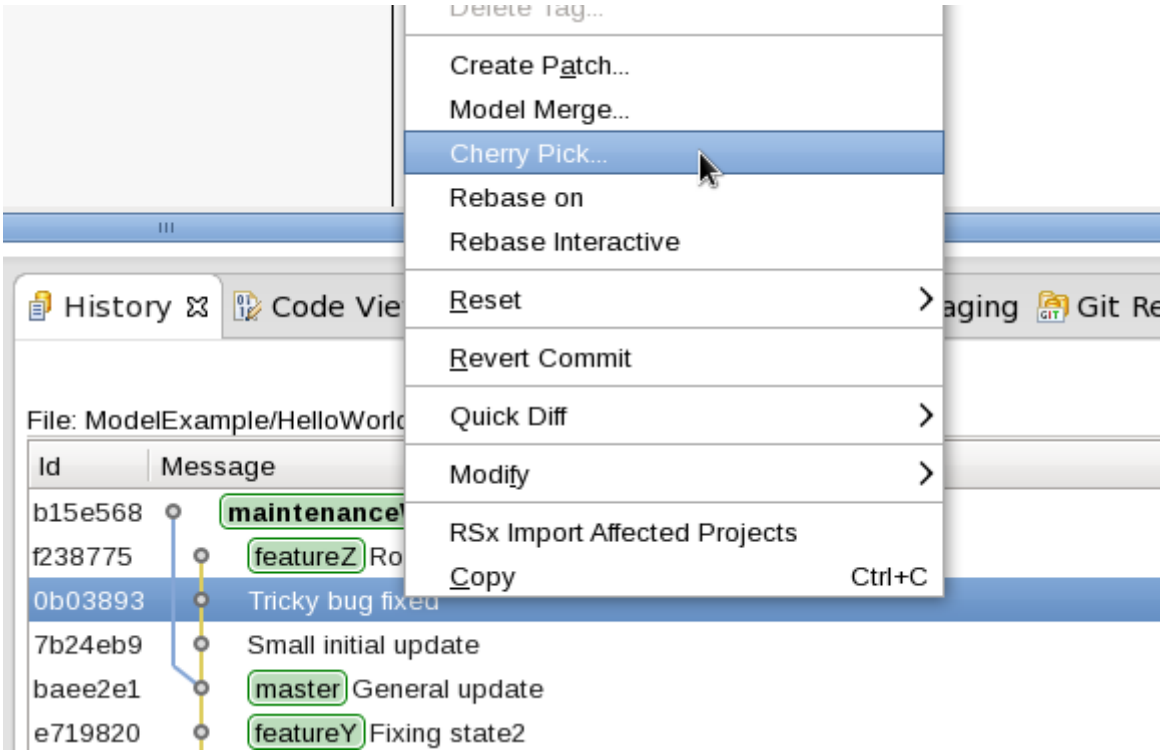
Merging a Specific Commit using Cherry Pick

In some situations you may want to add a specific change done as part of the work on some other branch to the branch you currently are working on without merging all changes done on the other branch. Using Git we can do this using the ”Cherry Pick” concept. In the RSARTE user interface this is most easily done from the History view.

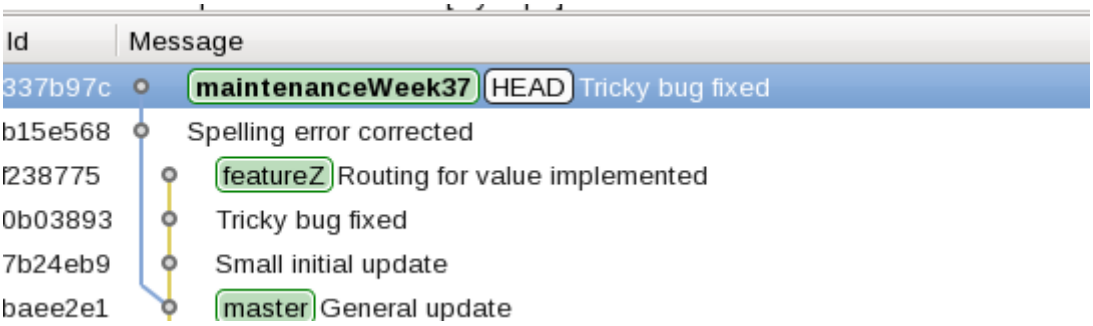
Assume we are working on the maintenanceWeek37 branch and have the following history.



We see that someone working on the ”featureZ” branch fixed a tricky bug and we would like to copy this fix to our own branch. From the History view we can easily do this using the ”Cherry Pick” command on the relevant node in the history.



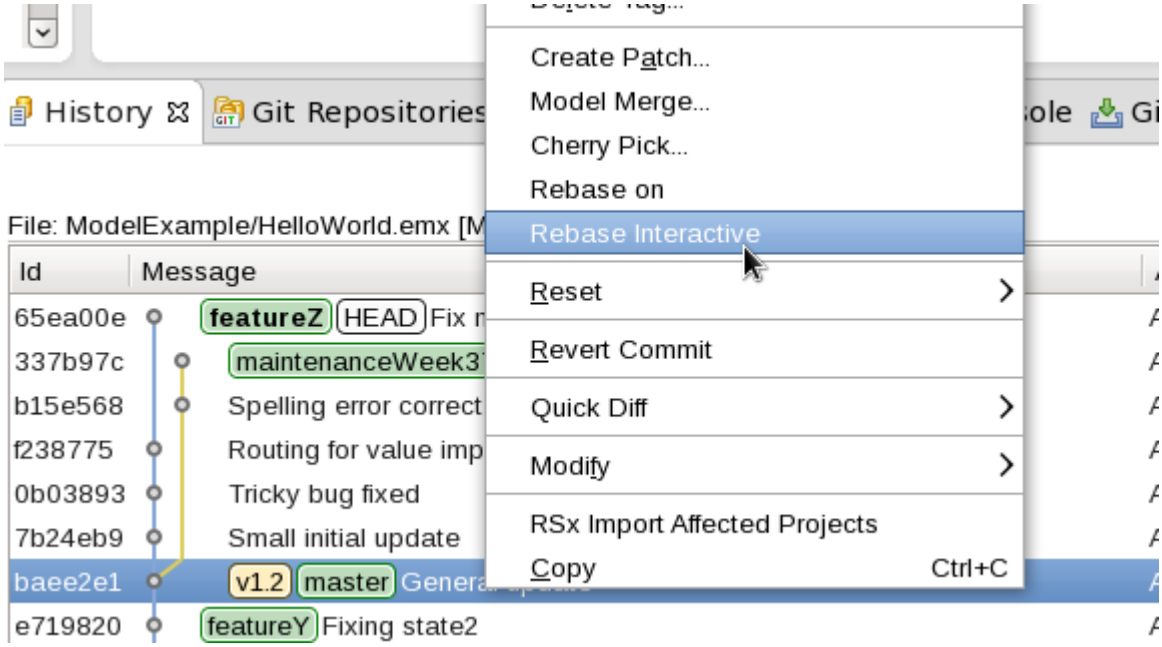
If accepting the default settings Git will now merge the specific changes associated with the ”Tricky bug fixed” commit and add them to our ”maintenanceWeek37” branch.



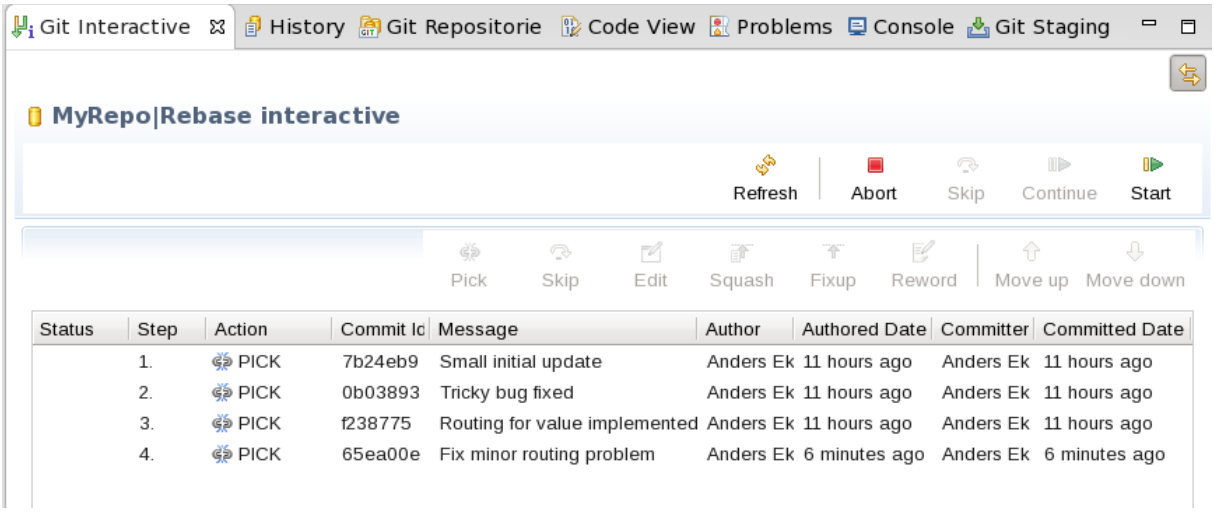
Interactive Rebase – Advanced Rewriting of History

Interactive Rebase is an advanced Git feature that combines a regular rebase with squash style merge and some aspects of cherry pick. You can use this as a convenient short-cut if you want to re-apply a series of commits (like in a regular rebase), but in the process also want to change the commit history (like a squash merge), selectively include or not include specific commits (slightly similar to a cherry pick) and even change some of the commits (like an amend commit operation).

You start a rebase by checking out the branch you want to modify. Then using for example the History view select the commit from where you want to change the commit history and select the ”Rebase Interactive” command.



In this example we would like to rewrite the commit history of the featureZ branch. The view that the ”Rebase Interactive” command provides gives us this possibility.


Git Interactive
History
Git Repositorie
Code View
Problems
Console
Git Staging
MyRepo|Rebase interactive
Refresh
Abort
Skip
Continue
Start
Pick
Skip
Edit
Squash
Fixup
Reword
Move up
Move down
Status Step Action Commit Id Message Author Authored Date Committer Committed Date
1. PICK 7b24eb9 Small initial update Anders Ek 11 hours ago Anders Ek 11 hours ago
2. PICK 0b03893 Tricky bug fixed Anders Ek 11 hours ago Anders Ek 11 hours ago
3. PICK f238775 Routing for value implemented Anders Ek 11 hours ago Anders Ek 11 hours ago
4. PICK 65ea00e Fix minor routing problem Anders Ek 6 minutes ago Anders Ek 6 minutes ago

The view will list all the commits we have selected and we can now determine what we want to do with each commit, before we start the rebase operation.

The defininition of the choice we make for each commit is shown in the ”Action” column. Initially the action is ”Pick” for all commits, but we can now change this. The possibilities we have are:

- Pick: Keep the commit as it is.
- Skip: Remove a commit
- Edit: Amend a commit
- Reword: Change the commit message
- Squash: Squash the commit with its predecessor

- Fixup: Same as squash, but ignore the commit message

It is also possible to re-order the commits using the ”Move Up”/”Move Down” buttons.

When you have finished defining the details of the actions to perform, you start the rebase operation with the ”Start” button.

For more details about interactive rebase take a look a the EGit user guide.

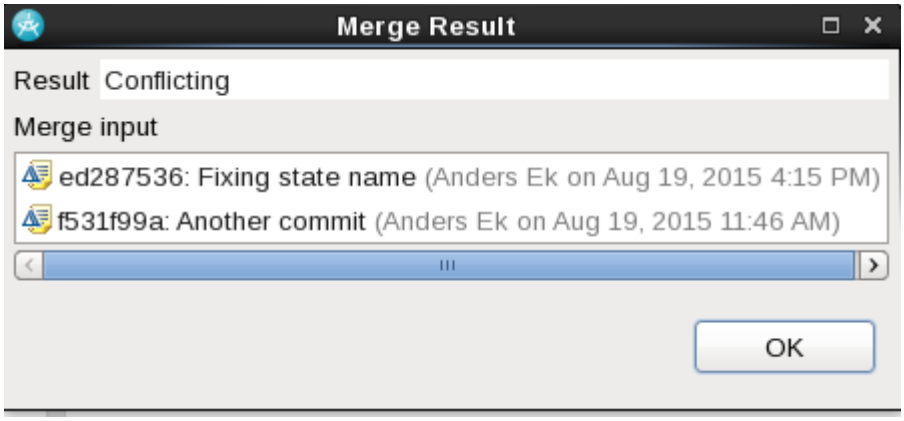
A word of warning: Avoid using interactive rebase on a branch that you have pushed to a remote repository and are sharing with other team members. This may cause serious problems for everybody that have downloaded the branch.

Resolving Conflicts

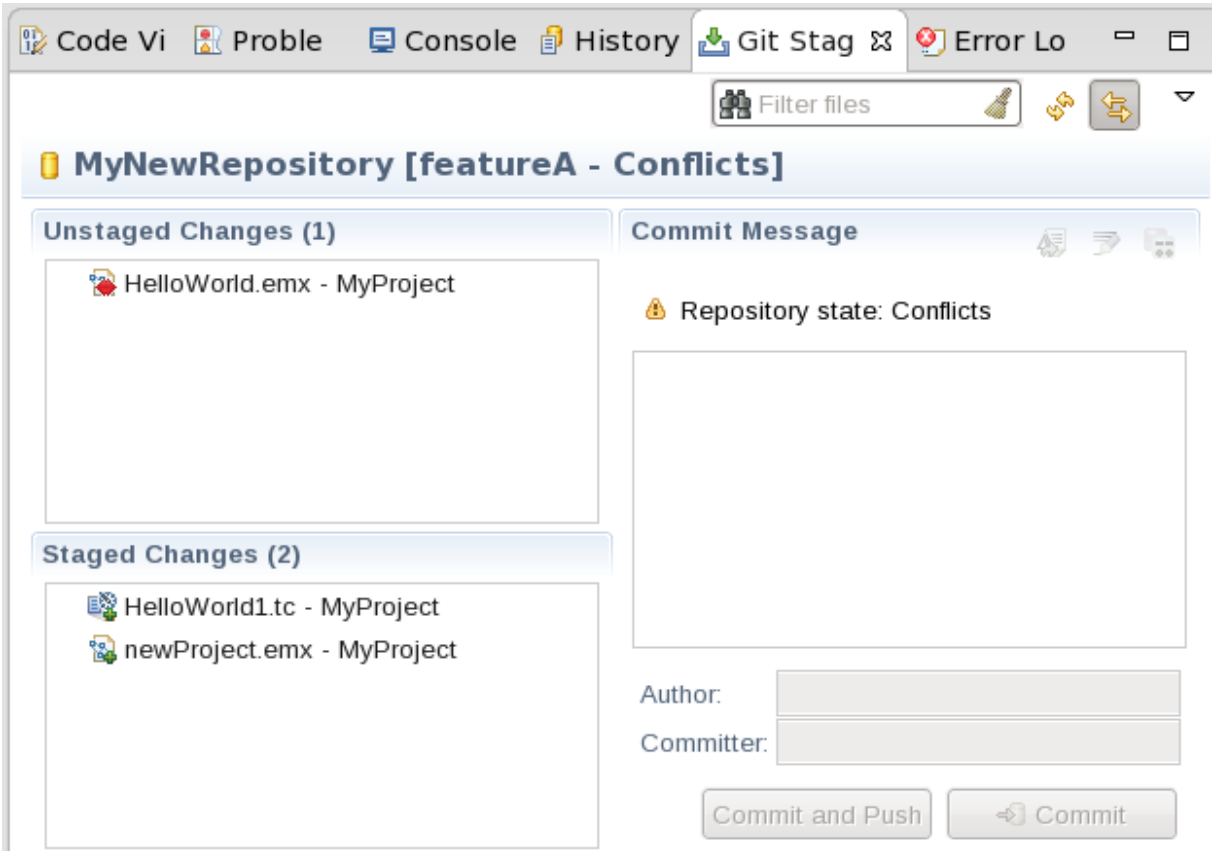
Unfortunately all merge and rebase operations are not as simple as the scenario we looked at in Merge and Rebase. The main complication is that the merge/rebase can result in a conflict. You will see this in the dialog that informs about the merge/rebase result.

Merge Conflicts

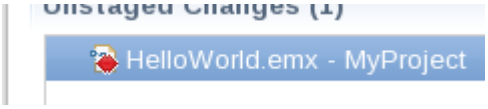
Let us start by looking at the merge scenario since this is slightly simpler and start by looking at the Invocation from the RSARTE user interface. When invoking a merge from e.g. the History view that triggers a conflict you will get the following dialog.


Merge Result
Result Conflicting
Merge input
ed287536: Fixing state name (Anders Ek on Aug 19, 2015 4:15 PM)
f531f99a: Another commit (Anders Ek on Aug 19, 2015 11:46 AM)
OK

The view that is used to understand and resolve the conflict is the Git Staging View. If you open the Git Staging View when the repository is in a conflict state you will see the text ”Repository state: Conflicts”. This is the Git Staging View indication that you need to resolve conflicts before committing your changes.

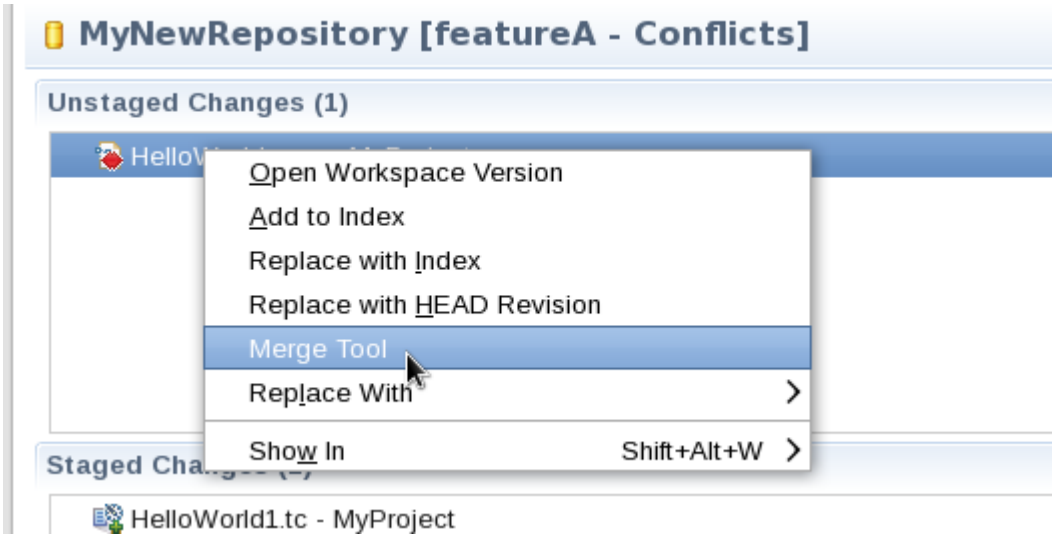


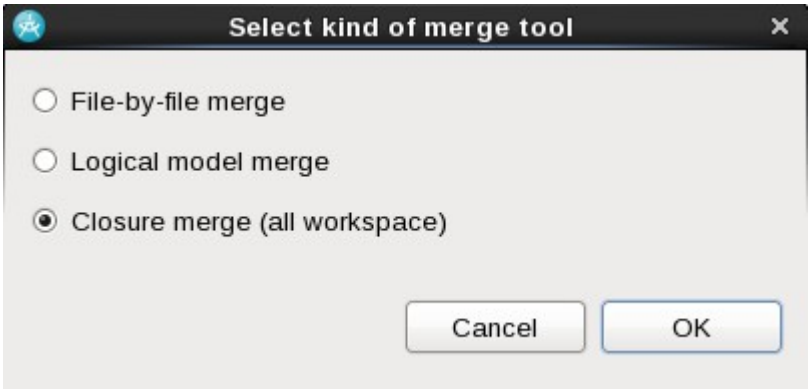
The conflicts that need to be resolved are marked with small red icons on elements in the ”Unstaged Changes” list (and actually also in most other views where the elements are shown).



The simplest way to resolve model conflicts is to use the Merge Tool in closure mode. This will result in fewer merge sessions and simpler difference descriptions than if using the other merge kinds.

To do this you can invoke the command ”Merge Tool” on any model file that is reported as in conflict in the list of ”Unstaged Changes” and choose ”Closure merge” (if using default settings).





This will invoke the standard interactive merge features of RSARTE as described in Comparing and Merging Models. After the merge session is saved the result can be committed to the Git repository. Note that all the model files will be added to the Git index as part of the interactive session.

The ”Merge Tool” can also be used to resolve conflicts for non-model files. If you invoke the command on this kind of file different interactive merge tools are invoked based on what kind of file it is.

If using command line Git the merge tool Invocation is normally done using the command ”git mergetool” or ”git mergetool <file>”. If you invoke the command without a file argument Git will automatically invoke a merge driver for each file that is in conflict.

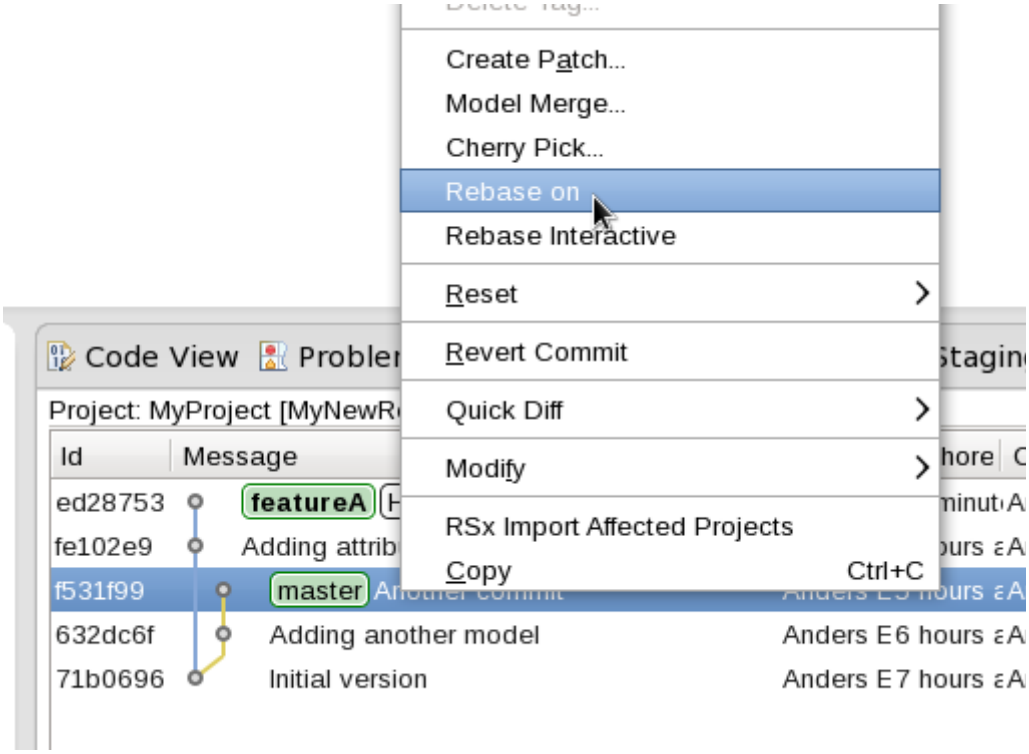
The same warning as for merge and rebase also applies to Git mergetool: **Do not use Git mergetool for model files if you do not have an integration with RSARTE that triggers RSARTE as the mergetool driver for model files.** This is likely to cause model corruption.

Due to the complexity of creating a merge tool integration with good performance that supports closure merge the currently recommended best practice is to use the RSARTE graphical user interface and the Git Staging View to resolve merge conflicts.

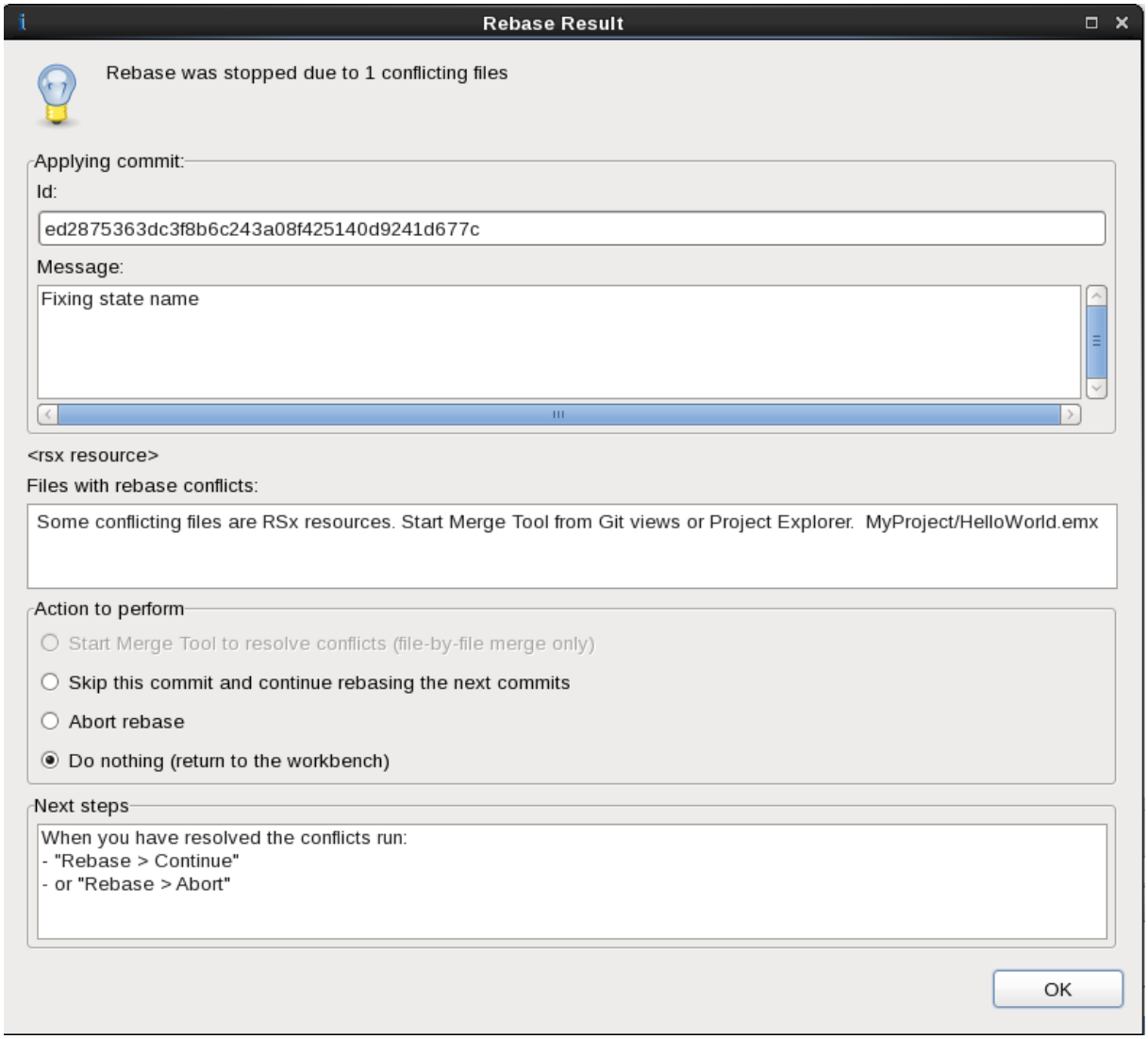
Rebase Conflicts

Since the Git rebase operation essentially is a sequence of merges the handling of conflicts in a rebase scenario is very similar to the merge scenario. The main difference is that after resolving conflicts found during a rebase you will need to continue the rebase instead of only committing the merge result.

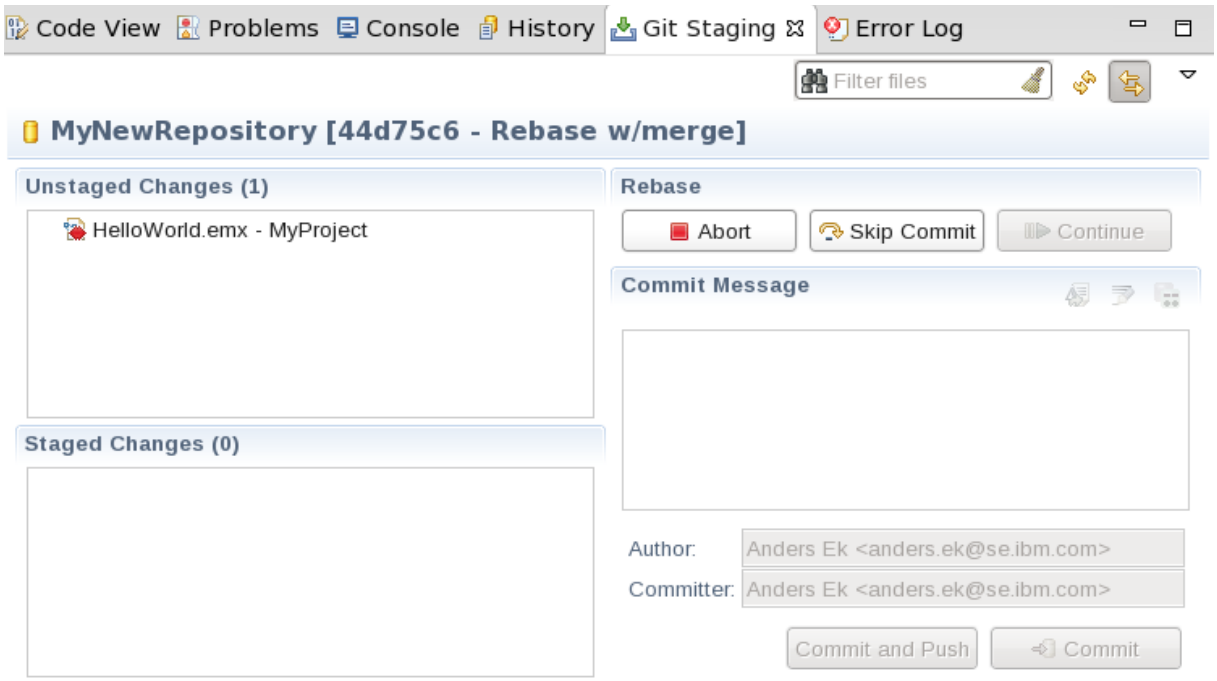
Assume starting in a situation where we have History view with a feature branch that contains conflicts w.r.t master branch. Now attempt to do a rebase on master in the same style as in section Merge and Rebase starting from the Git History view.



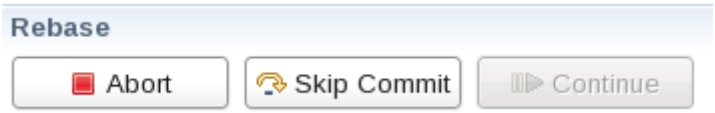
Instead of simple results dialog confirming that all went well you will now get a more complex dialog describing the rebase state.


Rebase Result
Rebase was stopped due to 1 conflicting files
Applying commit:
Id:
ed2875363dc3f8b6c243a08f425140d9241d677c
Message:
Fixing state name
<rsx resource>
Files with rebase conflicts:
Some conflicting files are RSx resources. Start Merge Tool from Git views or Project Explorer. MyProject/HelloWorld.emx
Action to perform
Start Merge Tool to resolve conflicts (file-by-file merge only)
Skip this commit and continue rebasing the next commits
Abort rebase
Do nothing (return to the workbench)
Next steps
When you have resolved the conflicts run:
- "Rebase > Continue"
- or "Rebase > Abort"
OK

The recommended method to handle this situation is to use the default choice ”Do nothing (continue to the workbench)” and again use the Git Staging View.

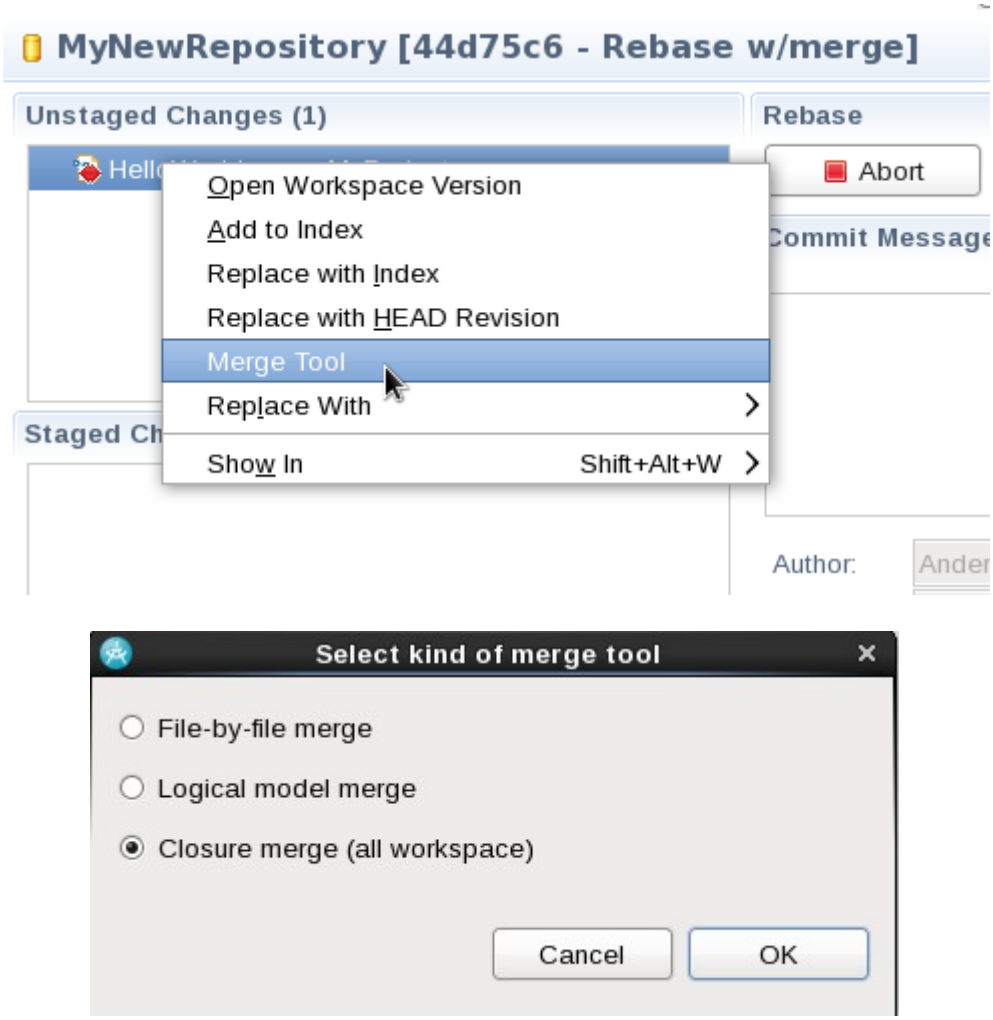


Note that the status shown now indicates that the repository is in a rebase state with buttons to Abort, Skip and Continue the rebase.



The Continue button is however not enabled, it will remain disabled until you have resolved all conflicts.

For model files the recommended way to resolve conflict is the same as for regular merge conflicts: Use the Merge Tool invoked on any model file in the ”Unstaged Changes” list using Closure Merge mode.



Now you can use the interactive merge features of RSARTE as described in Comparing and Merging Models to resolve the conflicts.

When you have resolved the conflicts you can use the ”Continue” button in the Git Staging View to continue the rebase operation. This button is available as soon as all conflicts are resolved. Continue the process of resolving conflicts using the Git Staging view until you get a dialog indicating that the rebase was successfully completed.

Working with Remote Repositories

One of the key ideas behind Git is that it is a distributed version control system. From a practical point of view the main implication is that you easily can work locally with your repositories and changes. Almost everything we have described in this document so far has been based on having local repositories. However, in order to share your work with the rest of the world and collaborate on joint efforts you need to extend the scope and also look at remote repositories.
When you work with remote repositories Git supports a multitude of different workflows, from a simple scheme suitable for a small team to complex schemes for large distributed projects. You can find a good description of some possible alternatives in Distributed Git -

Distributed Workflows. In addition, when using Git in complex projects it is commonly combined with supporting tools for managing the review of changes. A popular tool to manage reviews for Git repositories is Gerrit (https://www.gerritcodereview.com/). It is possible to combine RSARTE with Gerrit, both using the default Gerrit web based interface and using the review support in Mylyn (http://www.eclipse.org/mylyn/).

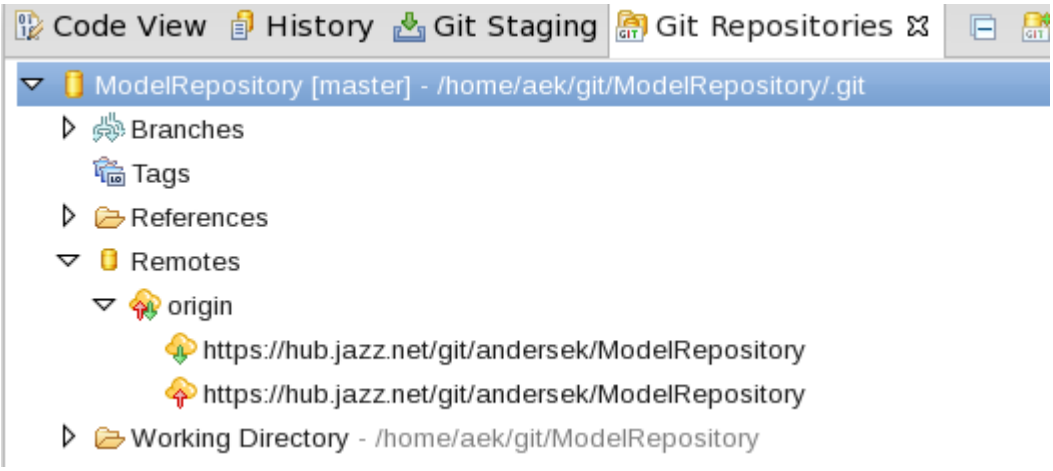
In this section and in Collaborating within a Team using Remote Branches we will focus on the simple scheme where you work directly with a remote Git repository, assuming that you have direct both read and write access to the remote repository.

A more advanced scheme is covered in Working with Remote Repositories using Gerrit Review Support where we focus on how to contribute your work to a shared repository managed using Gerrit.

Remote Repositories

If you followed the getting started instructions in Accessing a Remote Git Repository you most likely already have at least one remote repository configured. From the command line you can use the ”git remote -v” command to get information about what remote repositories are configured for your current local repository.

In the RSARTE GUI this information is most easily seen in the Git Repositories view. To see what remote repositories are configured expand the ”Remotes” node in the outline. This is what the Git Repositories view could look like if you have cloned a remote repository to get started.

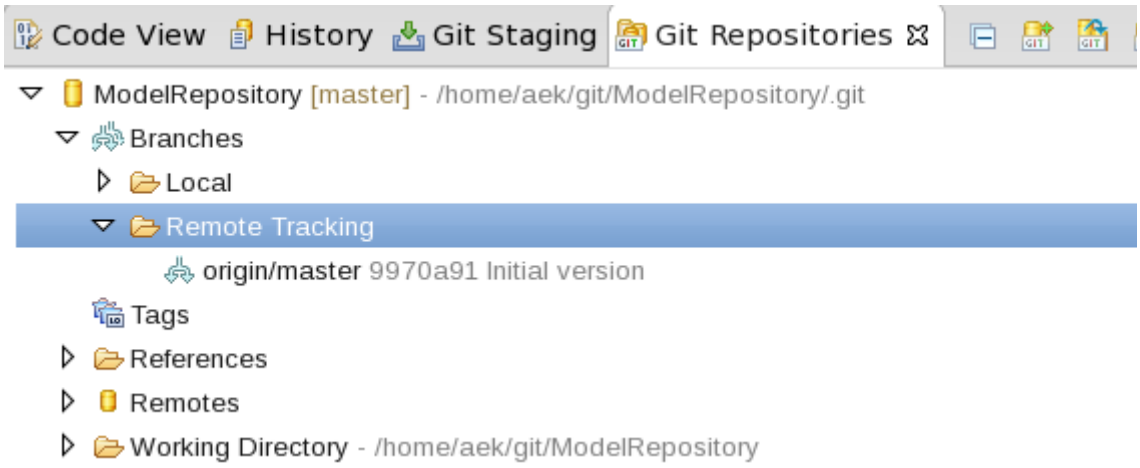


In this case you can see that we have configured one remote configuration, that is given the name ”origin” and that identifies a remote repository located at https://hub.jazz.net/get/andersek/ModelRepository.

Remote Tracking Branches

To ensure that local and remote repositories are not too tightly coupled Git introduces a special mechanism to locally keep track of the state of a remote repository. This is the concept of ”remote tracking branches”.

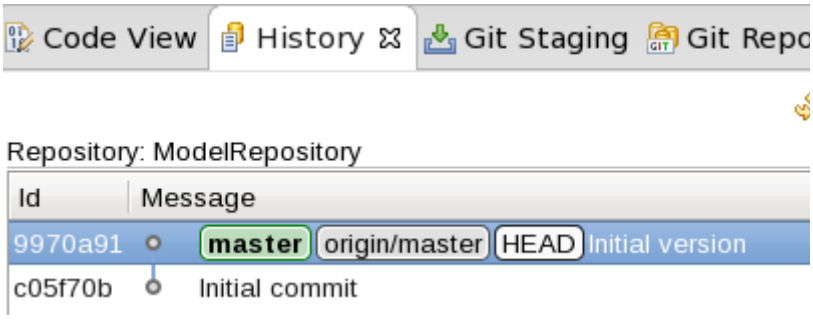
In our example above we have a remote repository called ”origin”. To track the state of this remote repository we will in our local repository have one or more branches only intended for this purpose. These branches can be seen in most of the relevant views in RSARTE. Let us first check the Git Repositories view. If you expand the ”Branches” node for your repository you will see that there now is another set of branches, shown in the ”Remote Tracking” part of the outline.



In our simple example we only have one remote repository named ”origin” and we can see that there is only one branch ”origin/master” that we track from this remote repository.

The remote tracking branches will be visible also in other views where branches are visible. For example the remote tracking branches are also visible in the History view.

This is what the simplest possible example can look like. This is the history of a newly created repository with one ”origin/master” remote tracking branch (and one local ”master” branch).

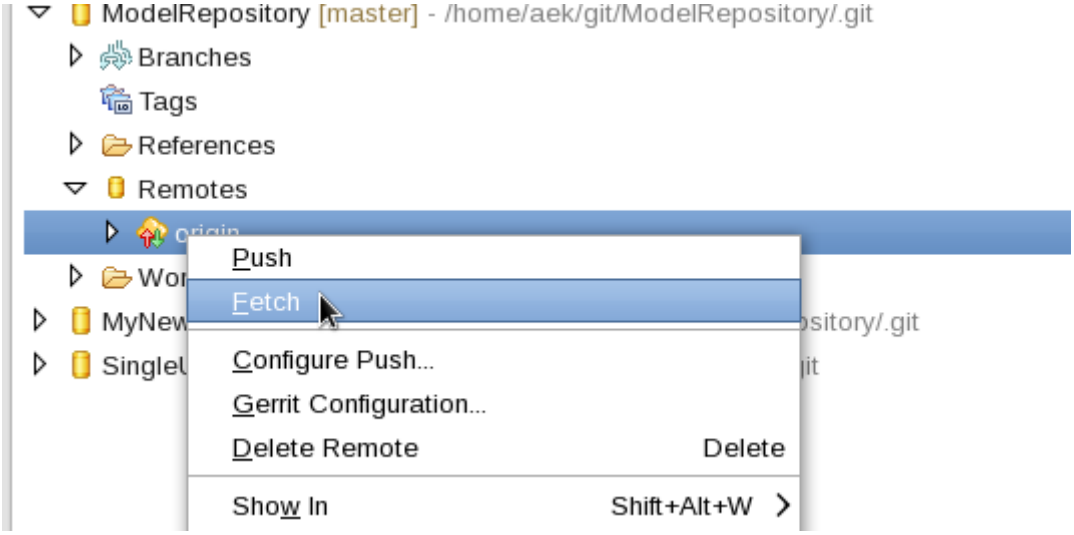


Note that the remote tracking branches are read-only from a local point of view. They will only be updated when you use various commands to synchronize the state of your local repository with the remote repository.

Update from Remote Repositories: Fetch

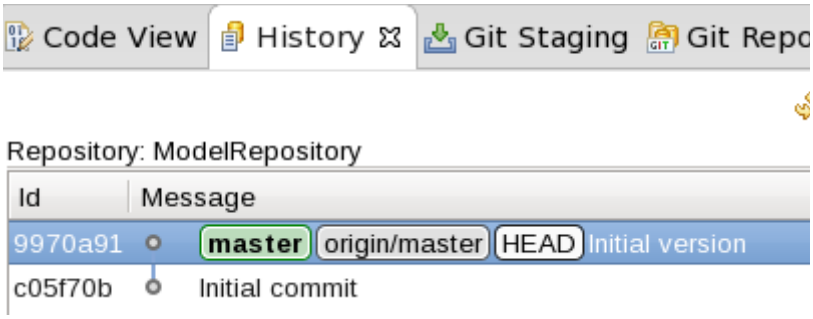
It is easy to update the local repository with changes done in the remote repository. The most basic way of doing this is using the ”Fetch” command. In our simple example from previous section the command ”git fetch origin” would retrieve any changes done in the remote repository called ”origin” and make them available in the remote tracking branch ”origin/master”.

In the EGit GUI the ”Fetch” command is of course also available, for example if you select a remote repository in the Git Repositories view.

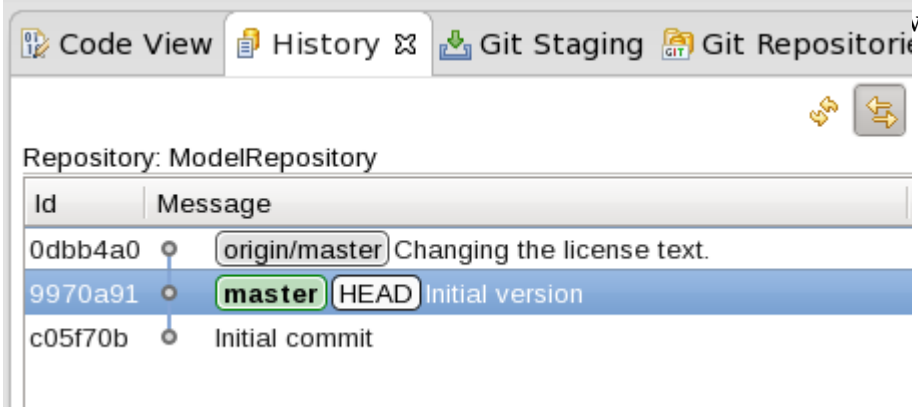


Note that the only changes done to your local repository by the Fetch command is to the remote tracking branches. No local branches are updated.

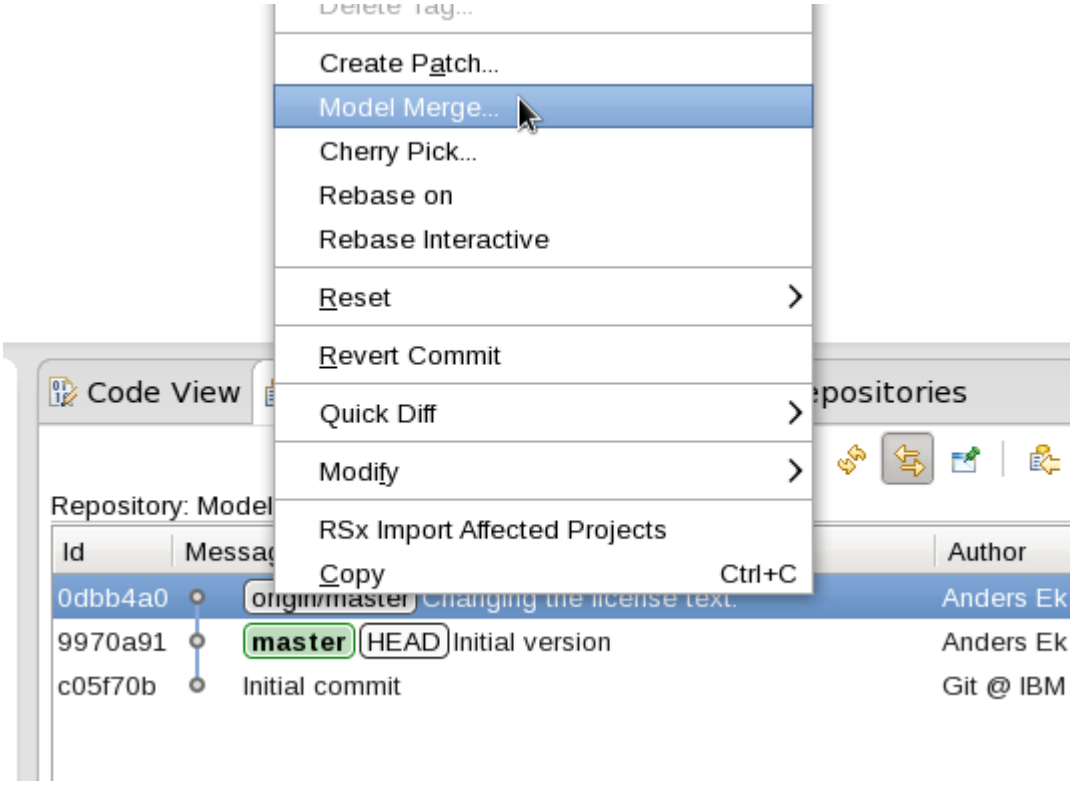
Assume for example that someone has updated the remote repository we called ”origin” and we do a Fetch command. The history before the Fetch:



History after Fetch:

As you can  what we

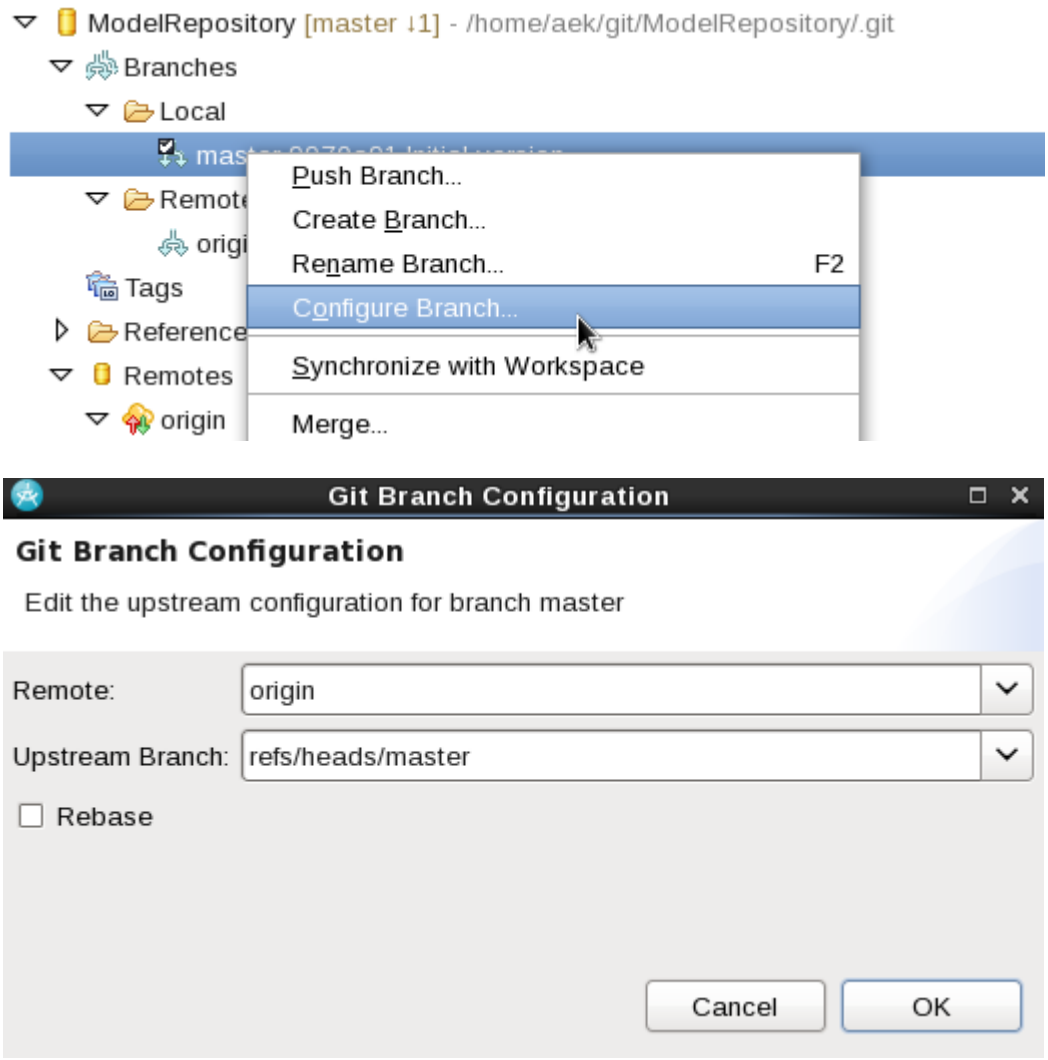
In most cases we now want to merge this change into our own local ”master” branch. This can be done exactly as for any other branch using the ”Model Merge...” command.



Upstream Configurations – Local Branches vs. Remote Tracking Branches

To make it easier to incorporate changes from remote tracking branches to our local branches (and vice versa) it is possible to define an ”Upstream Configuration” for a local branch. If you have used ”clone” to get a copy of a remote repository this is actually most likely already done for you.

To check and update the upstream configuration for a local branch we can use the command ”Configure Branch” from the Git Repositories view.



In this case we can see that the local ”master” branch is defined to have an upstream remote tracking branch that is the ”master” branch on the ”origin” remote server. (You can ignore the ”refs/heads/” part of the name for now.)

In practice setting up the upstream configuration is what enables us to use two commands to update our local repository from the remote and vice versa:

- The Pull command will fetch the latest changes and automatically merge them based on our upstream configuration and the remote tracking branch configuration.
- The Push command will do the opposite. It will take our latest local changes, send them to the remote repository, commit them and fetch back the changes to the remote tracking branch. All based on our upstream and remote tracking branch configurations.

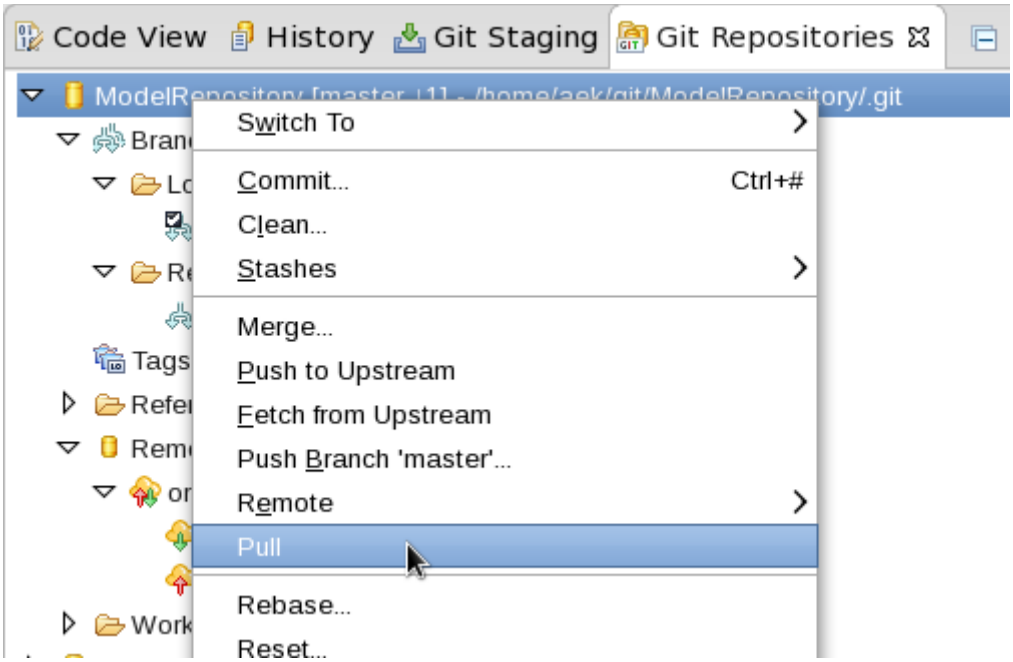
Update from Remote Repositories: Pull

If you have an upstream configuration for a branch that identifies a remote tracking branch, and thus also identifies a remote repository you can instead of doing a Fetch + Merge as in section Update from Remote Repositories: Fetch invoke a Pull command. This will first do a

Fetch from the remote repository to retrieve the remote changes and then automatically merge the changes. Note that the Pull command is based on the branch that is currently checked out and will change the files you currently have in the workspace similar to what the corresponding merge would have done.

In general it is safer to use fetch + merge instead of pull, but if you know what you are doing it is a convenient short-cut.

You can invoke the Pull command for example from the repositories view on any local repository.

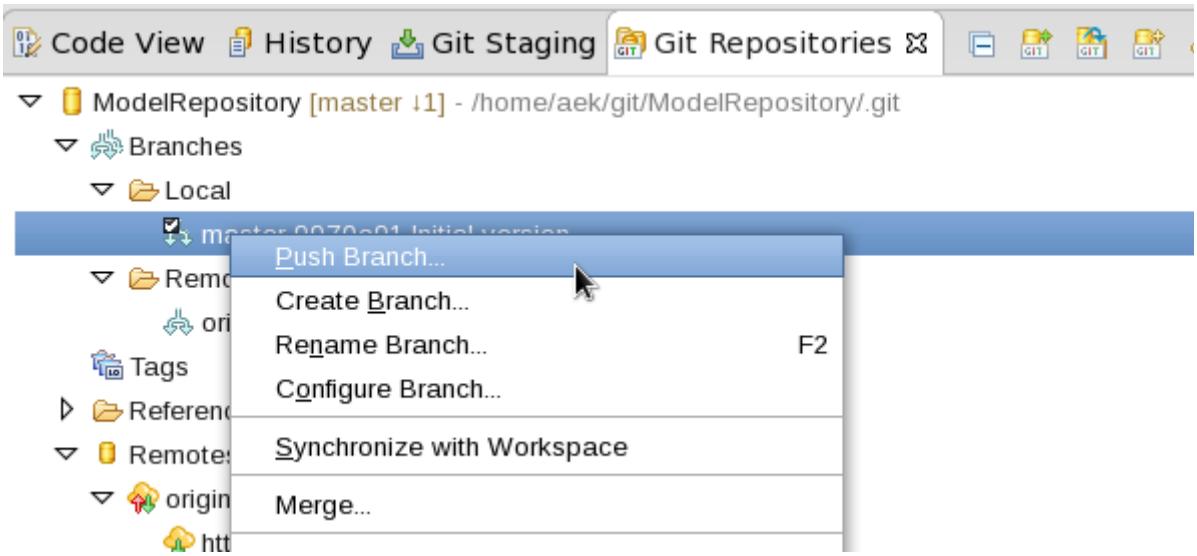


Updating the Remote Repository with Your Local Changes using Push

To be able to push your local changes in a simple fashion to a remote repository you will need to have both a remote tracking branch and an upstream configuration as discussed in sections Remote Tracking Branches respectively Upstream Configurations – Local Branches vs. Remote Tracking Branches set up. There are other, more advanced ways to determine what and how to push, see e.g. The EGit Tutorial or the Git Reference for details. However the most convenient mechanism is based on using remote tracking branches and the upstream configuration to handle the mapping and that is what we'll use now.

You can invoke the Push command from the command line as ”git push origin master”. ”origin” here indicates the remote repository and ”master” the branch to push.

From the RSARTE user interface a simple way to do a Push is to start from the Git Repositories view. Identify the branch you want to push and use the command available in the context menu.



Collaborating within a Team using Remote Branches

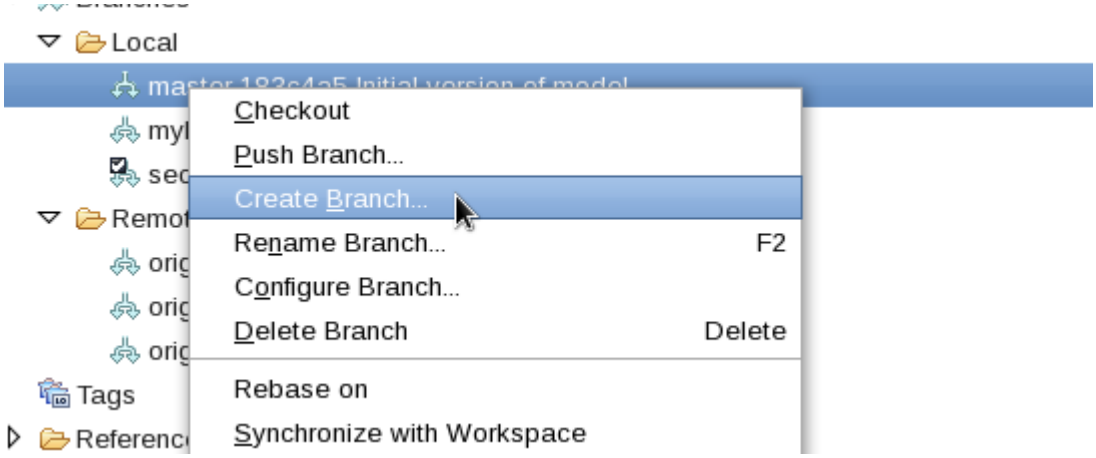
So far in the remote repository discussion we have used the ”master” branch in all examples. This is enough if there is only one team working on a project. All team members have write access to the ”master” branch and they use this branch to share the progress of the work. However, as soon as there is more than one team working on a project the Git remote branch concept is the recommended way for the team members to share work. A remote branch is simply any branch that is available on a remote server that all team members have access to. Using one or more specific branches for the team's work they can use the remote server to share progress while still not disturb the rest of the organization.

Creating a Remote Branch

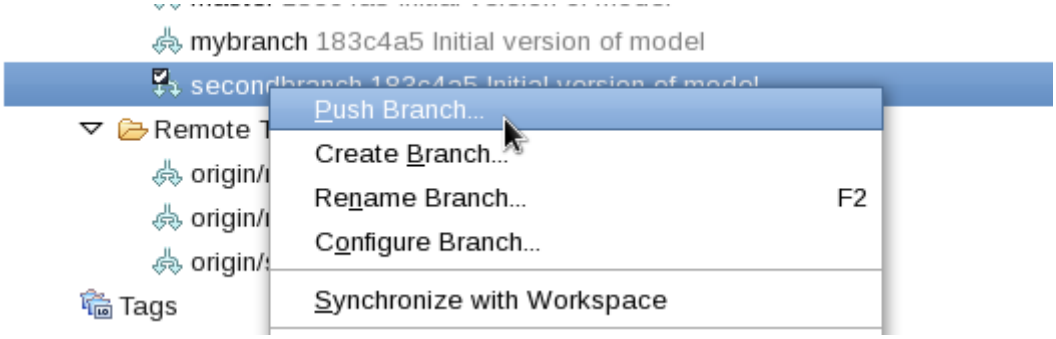
To create a remote branch based on current master branch using command line Git two commands are used:

- ”git checkout -b mybranch master” to create the branch locally
- ”git push -u origin HEAD:mybranch” to push the newly created branch to the remote repository called ”origin”, giving it the same name as the local branch. The ”-u” makes Git configure a remote tracking branch and upstream configuration automatically.

You can do the same operation using the EGit commands in the RSARTE user interface. Create the branch as usual using for example the ”Create branch...” command from the Git Repository view.



Then push the new branch to the remote repository.



Configuration of and Basic Operations for Remote Branches

Fortunately working with a remote team branch is from most points of view not different than working with the master branch as we discussed in section Working with Remote Repositories. You can configure the remote tracking branch and the upstream configuration the same way, except of course specifying the name of the team branch instead of ”master” in the relevant configuration dialogs.

When the configuration is done you can then use the Fetch, Pull and Push operations to update your local repository from the common repository and vice versa.

Working with Remote Repositories using Gerrit Review Support

In sections Working with Remote Repositories and Configuration of and Basic Operations for Remote Team Branches we looked at a simple scheme where everybody can both read and write from a common repository without constraints. This works well for a small organization with a limited number of teams working on a reasonably small common project. However, when the size grows and the complexity of the application increases a scheme that adds some

constraints on the process of pushing contents to the master branch in the repository is useful to avoid problems.

Git supports many ways to organize the workflow to suit different teams and organizations, see for example Distributed Git - Distributed Workflows for an introduction. However, the common idea is that there is one main common repository that represents the official version of the application source code. Then there is a scheme put in place that guards who and how changes can be pushed to this repository.

One common scheme that we will look at in this section is to use Gerrit as the guardian (https://www.gerritcodereview.com/). The details of the process that guards the master branch is configurable and is outside the scope of this document. Fortunately the overall workflow as seen from our RSARTE developer's point of view is similar for most scenarios.

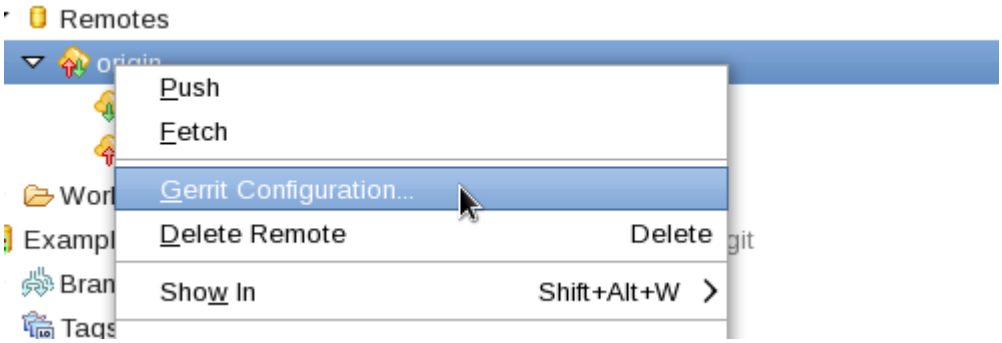
Cloning and Initiating a Repository from Gerrit

When using Gerrit the common main repository is usually hosted at the Gerrit server. Getting access to a repository from a Gerrit server is no different than from any other remote server. So the process is the same as is described in section Accessing a Remote Git Repository The only aspect to be aware of is that in most common Gerrit installation configuration you will need to have ssh as the protocol used to access the repository. It is possible to change after cloning, but easiest to handle already from start.

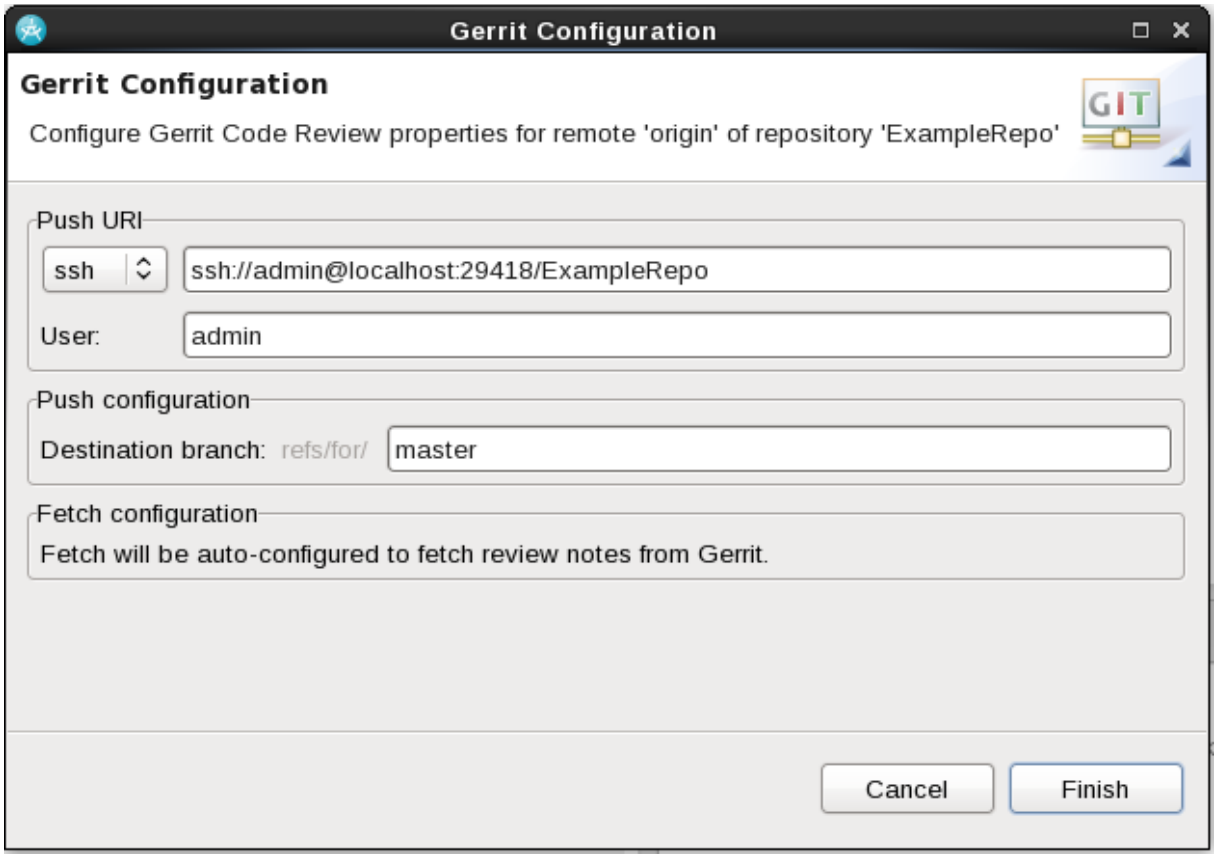
When you have cloned the repository you will need to tell RSARTE that you want to enable Gerrit specific commands for the repository and identify the remote that is to be used for the Gerrit specific commands.

In some cases the Gerrit configuration will be set up automatically. You can check this by looking at the icon for the repository in the Git Repositories view that if configured for Gerrit will contain a small green and red indicator.

In any case you can check and change the configuration using the ”Gerrit Configuration” command available for remotes in the Git Repositories view.



The Gerrit configuration makes it possible both to set up the access method and remote address of the Gerrit server and the review branch in the Gerrit repository you will push review requests to.



When the repository is enabled for Gerrit you will find some extra features, in particular the ”Push to Gerrit” and ”Fetch from Gerrit” commands available for models in the Project explorer and repositories in the Git Repositories View.

Gerrit Basics

The purpose of Gerrit is to enable review of commits. To accomplish this two key concepts are used:

- ”change ids”: A unique identifier that identifies a review request in Gerrit
- ”for” branches: Branches in the Gerrit hosted repository that contains the commits to be reviewed.

From a pragmatic RSARTE Git integration point of view the change ids will show up in the end of commit messages. The recommended practice is to use a Git commit hook that automatically will add the change id when you commit a change in RSARTE. This can be downloaded from the Git repository and we will assume that you use this hook. The real-life result is that if you check your commits e.g. in the History view you will see lines starting with ”Change-Id:” and followed by long identifier added to all your commit messages.

The ”for” branches will show up when you push changes to Gerrit. In Updating the Remote Repository with Your Local Changes using Push we used an Upstream Configuration to identify a Remote Tracking Branch and used this to identify the branch in the remote server we wanted to push our changes to. We also used the same configuration for both fetch and push. When using Gerrit this is no longer the case. We still use a remote tracking branch for fetching changes, but when pushing we instead push to a special ”for” branch on the Gerrit server. The idea is that someone using Gerrit will review our changes (and potentially do other checks, like building and running tests) and if all is fine submit the change we proposed. The change will then automatically be merged to the ”real” branch and we can retrieve the new version using a fetch. We will take a close look at the push and review support later in this chapter.

Basic Operations when using a Gerrit Based Repository

Before entering the specific workflows concerning push and review in a Gerrit context it is worthwhile to consider other operations you typically would do in RSARTE w.r.t. Git repositories. Fortunately most operations will work the same since a Gerrit server is from most points of view no different than any other Git remote server.

Since you still work in your local repository all the key workflows with respect to understanding the version history, investigating commits and committing your local changes are the same. The only exception is the ”change ids” that will appear in commit messages.

Also fetching changes from remote repositories and working with a team on shared remote branches follows the same principles. The exception being of course the review process that will change the way you push your changes. It is worth noting though that depending on how your Gerrit server is configured you can potentially also continue to push as described previously in this the document, not starting a review process, as an alternative to the Gerrit review triggering push. All depending on the policy used by your organization to handle contributions to the common main repository maintained by the Gerrit server.

Push for Review

When you want to push a commit you have created for review using Gerrit you will as mentioned before push to a special branch identified by a ”for” prefix. In most cases the branch identification will look something like ”refs/for/mybranch” instead of the usual ”refs/head/mybranch” that we have seen before.

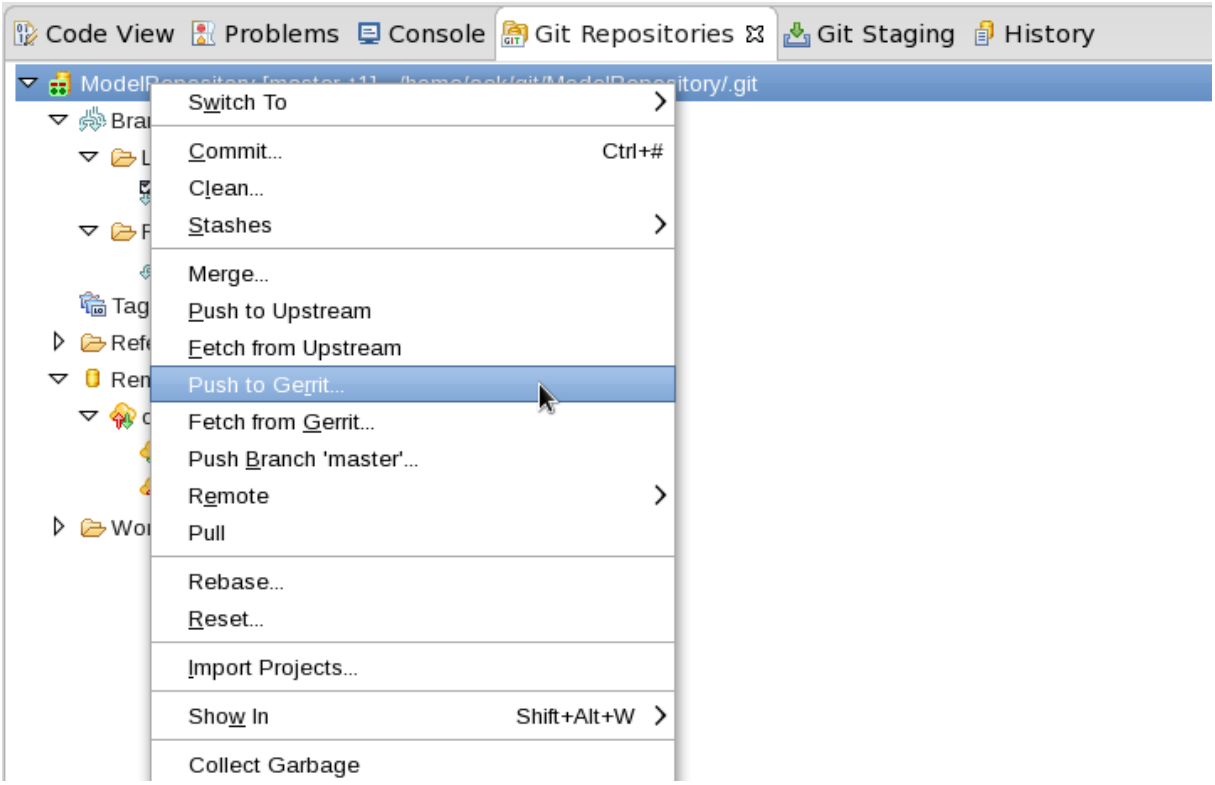
If doing this from command line you would use a command similar to the following:

```
git push HEAD:refs/for/mybranch
```

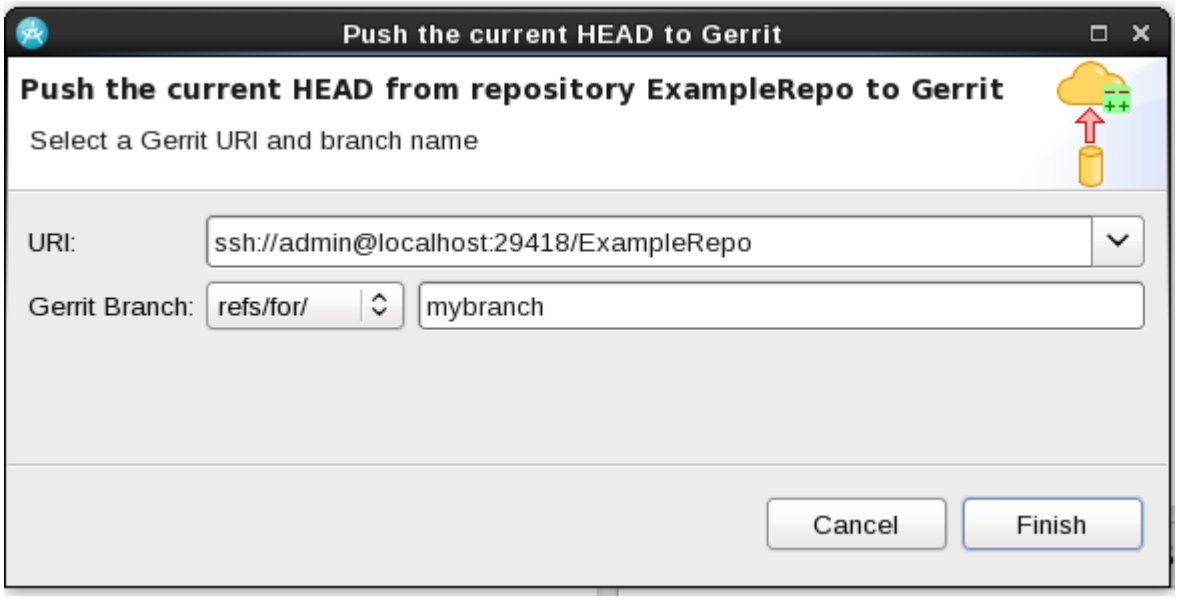
This particular command assumes that:

- ”mybranch” is the name of the branch you want to push to, and
- you have the corresponding local branch checked out so HEAD identifies the commit you want to push.

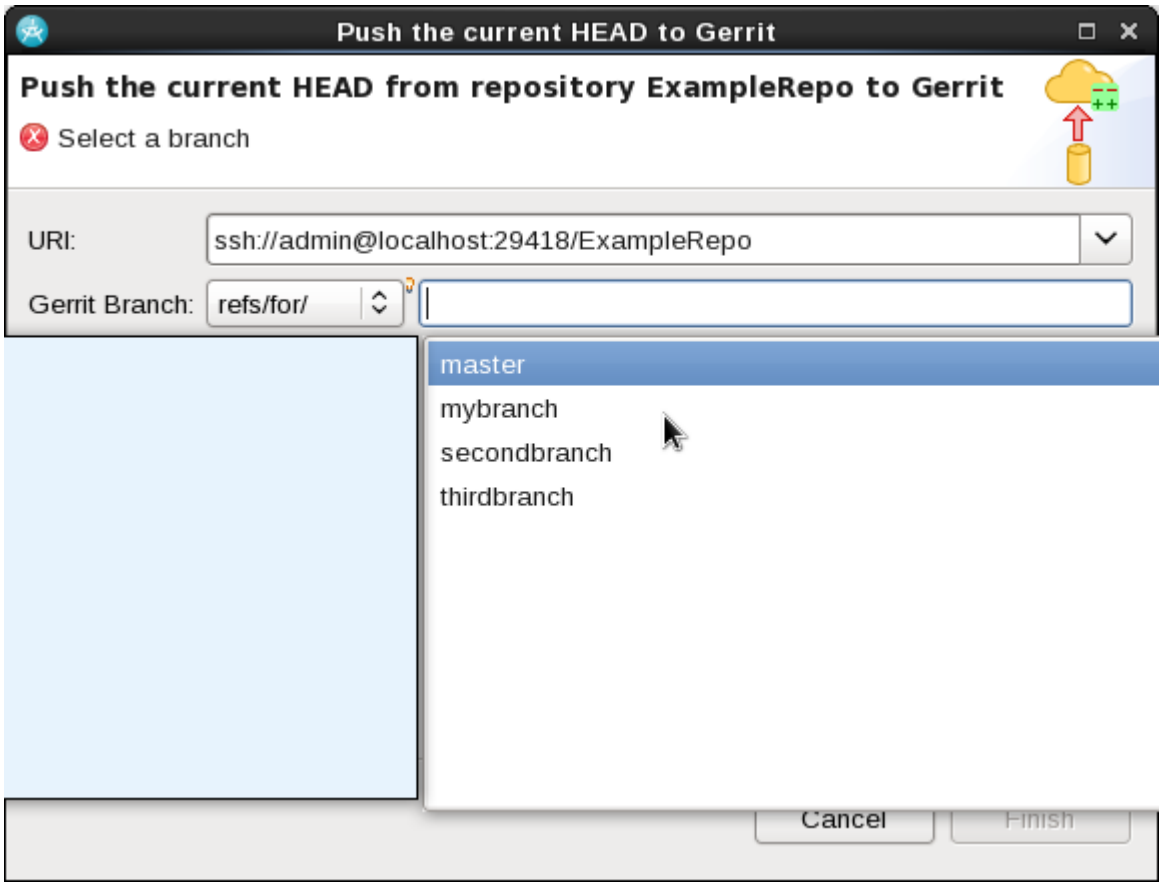
If using the RSARTE user interface you will have a ”Push to Gerrit...” command available for repositories in the Git Repositories view. Also in this case you will need to have the local branch checked out so HEAD identifies the commit that you want to push for review.



This will pop a dialog where you can specify the push details.



It is worth noting that there is a name completion help in the ”Gerrit Branch” field. If you press Control-Space in this field you can choose between the currently available branches.



In any case the end result is that your commit now will be available in the Gerrit server for review.

Perform a Review using Mylyn

Mylyn (http://www.eclipse.org/mylyn/) is a task and application lifecycle management framework for Eclipse. From an RSARTE point of view this has one very useful property: It enables reviewing Gerrit changes directly from inside RSARTE. This is especially convenient when using a modeling tool like RSARTE where comparing two model files using the text based compare methods available in the Gerrit web interface is very inconvenient.

Mylyn contains many additional features and you can find a good description of both the review support and other features in the Mylyn User Guide. In this section we will only look at the basic steps you are likely to need when reviewing Gerrit change requests.

Installing and Configuring Mylyn

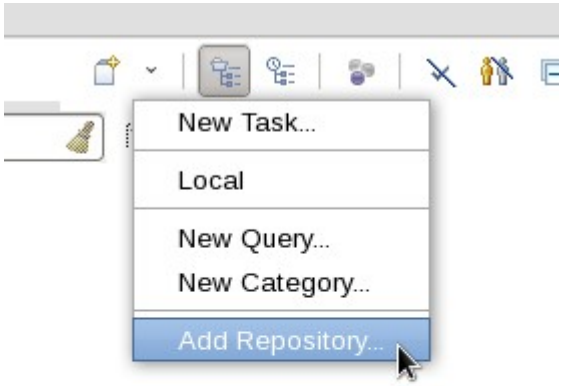
If Mylyn is not installed by default in you environment you can download it from http://www.eclipse.org/mylyn/ and install it using ”Help->Install New Software” in RSARTE.

When installing it is recommended to install at least the following features:

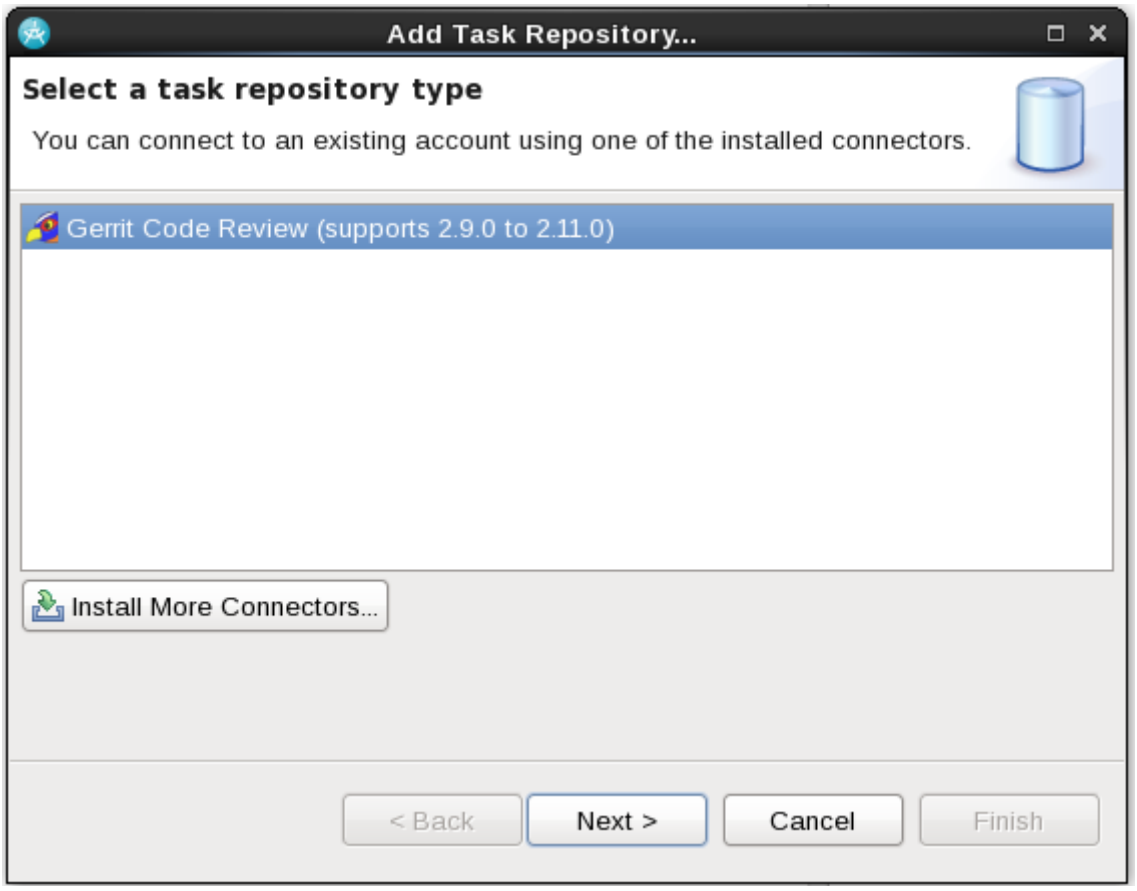
- "Mylyn Task List"
- "Mylyn Task-Focused Interface"
- "Mylyn Reviews Connector: Gerrit"
- "Mylyn Reviews Connector: Gerrit Dashboard"

Before you can use Mylyn for review you also need to connect it to your Gerrit server. This can be done from the ”Task List” view of Mylyn. A convenient way is to open the ”Planning” perspective in Eclipse. This is also a perspective useful for reviewing changes as we will see in next section.

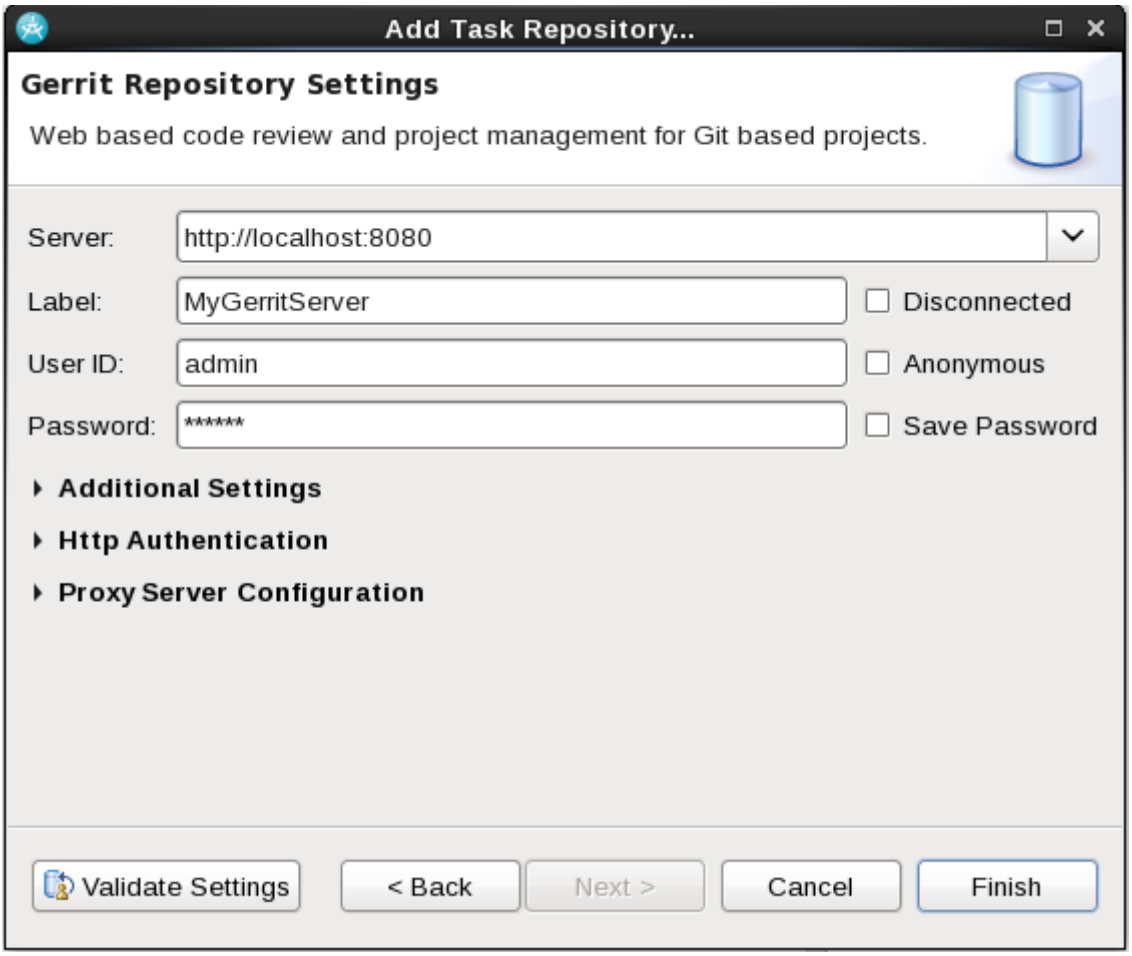
In the Task list you can use the ”Add Repository...” command connect to Gerrit.



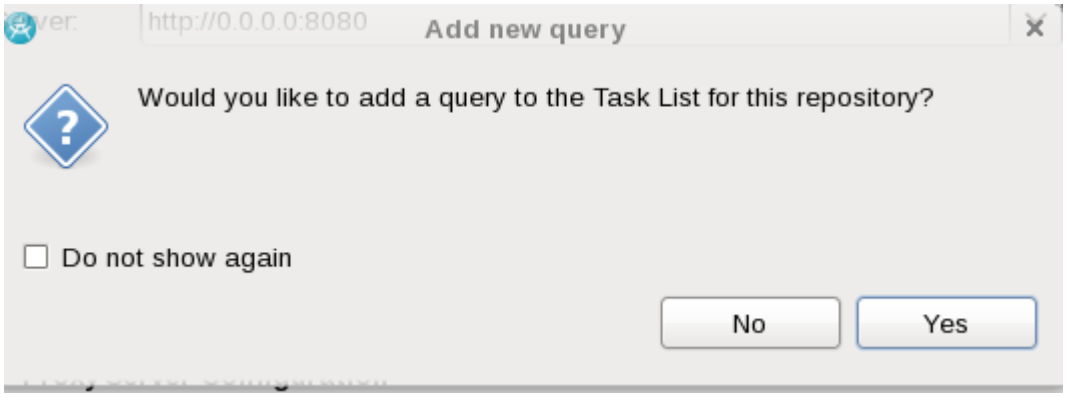
Choose the Gerrit connection if you have more than one Mylyn connection installed.


Add Task Repository...
Select a task repository type
You can connect to an existing account using one of the installed connectors.
Gerrit Code Review (supports 2.9.0 to 2.11.0)
Install More Connectors...
< Back
Next >
Cancel
Finish

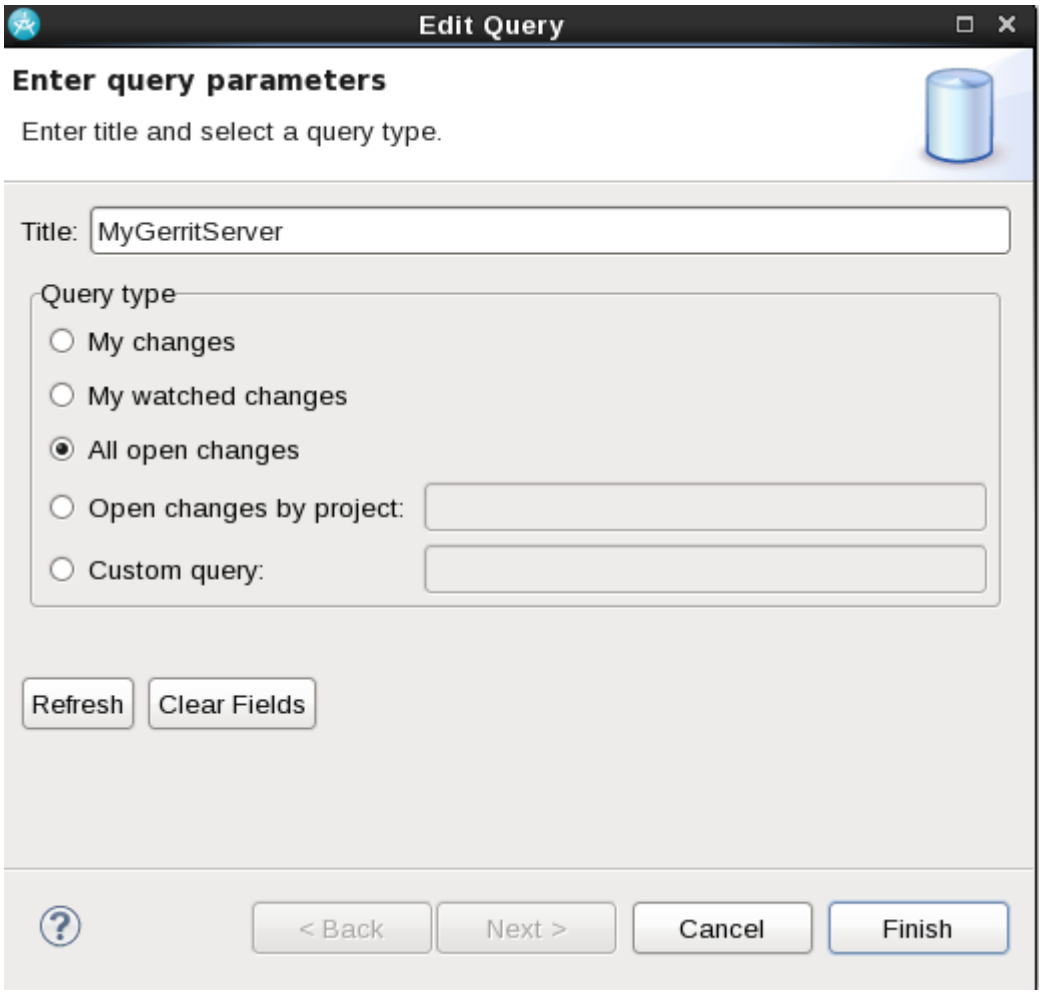
You will need to specify the URL of your Gerrit server and your login credentials in following dialogs.



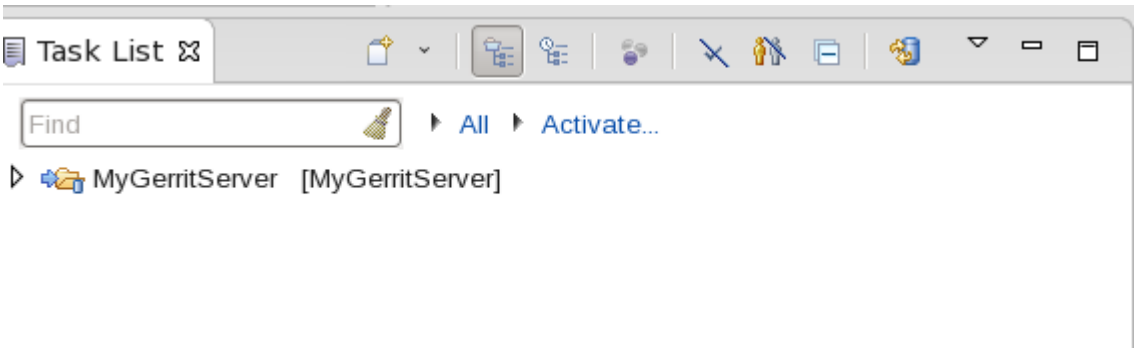
It also makes sense to choose to add a query automatically when prompted.



You can select the scope of what you want to see. Only your own requests or all?

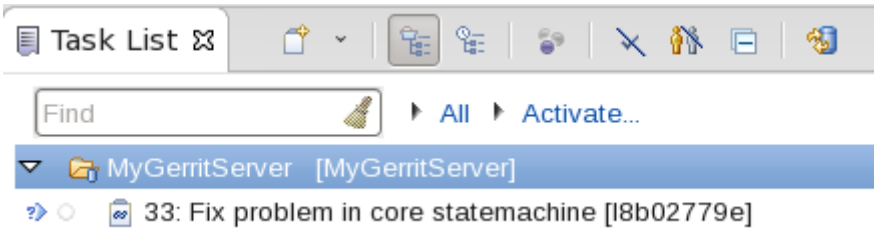


When done you will have a Task List view that is configured to show review requests from your Gerrit server.



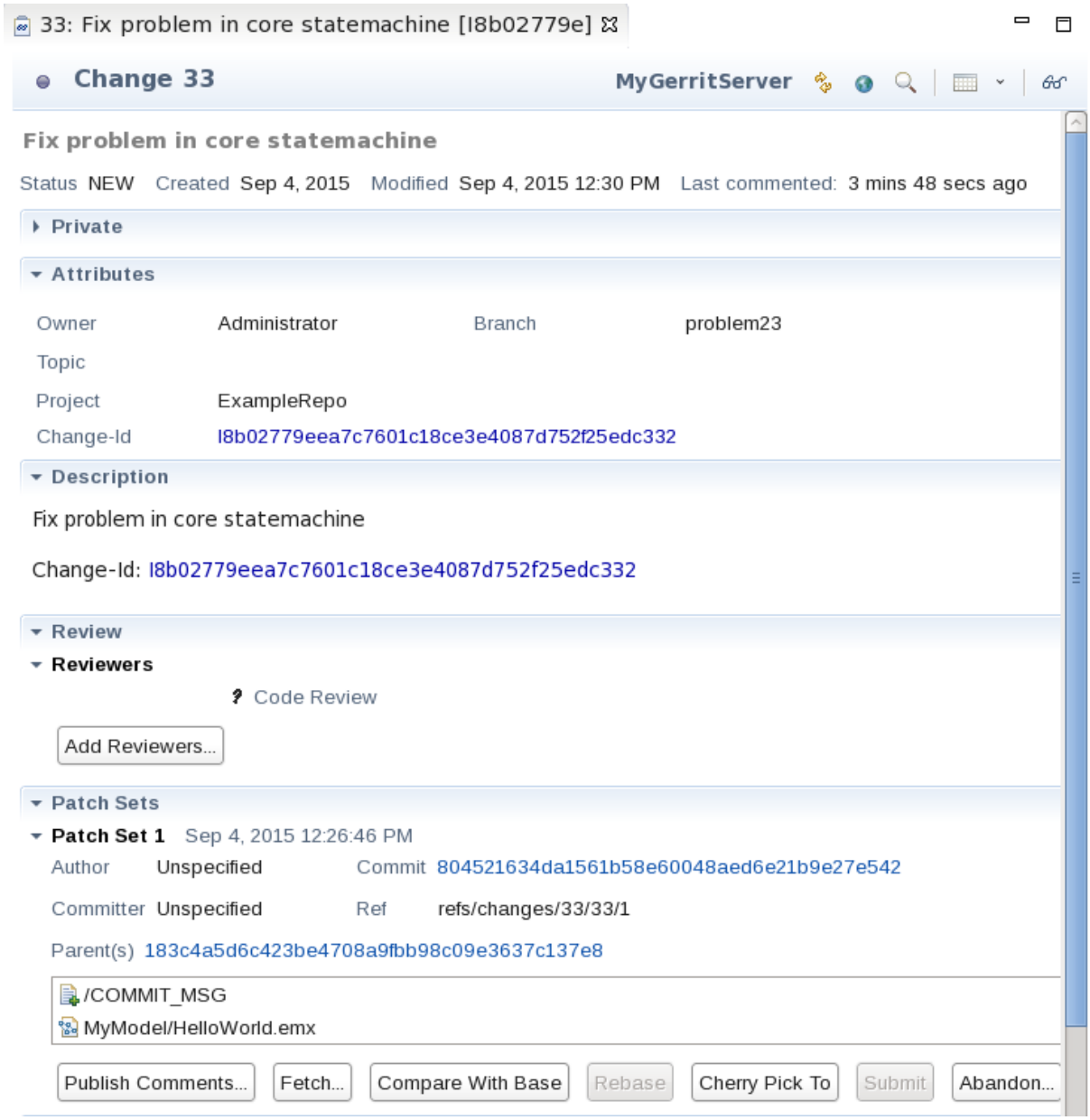
Reviewing a Change Request in Mylyn

When you want to review the changes assigned to you (or other changes) the first step is to look at the ”Task List” view again. The changes that match the query you created will now be visible in the query node in the list.

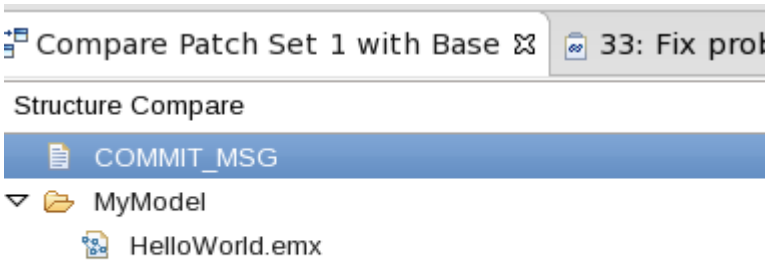


If you don't see what you expected press ”F5” to refresh the list.

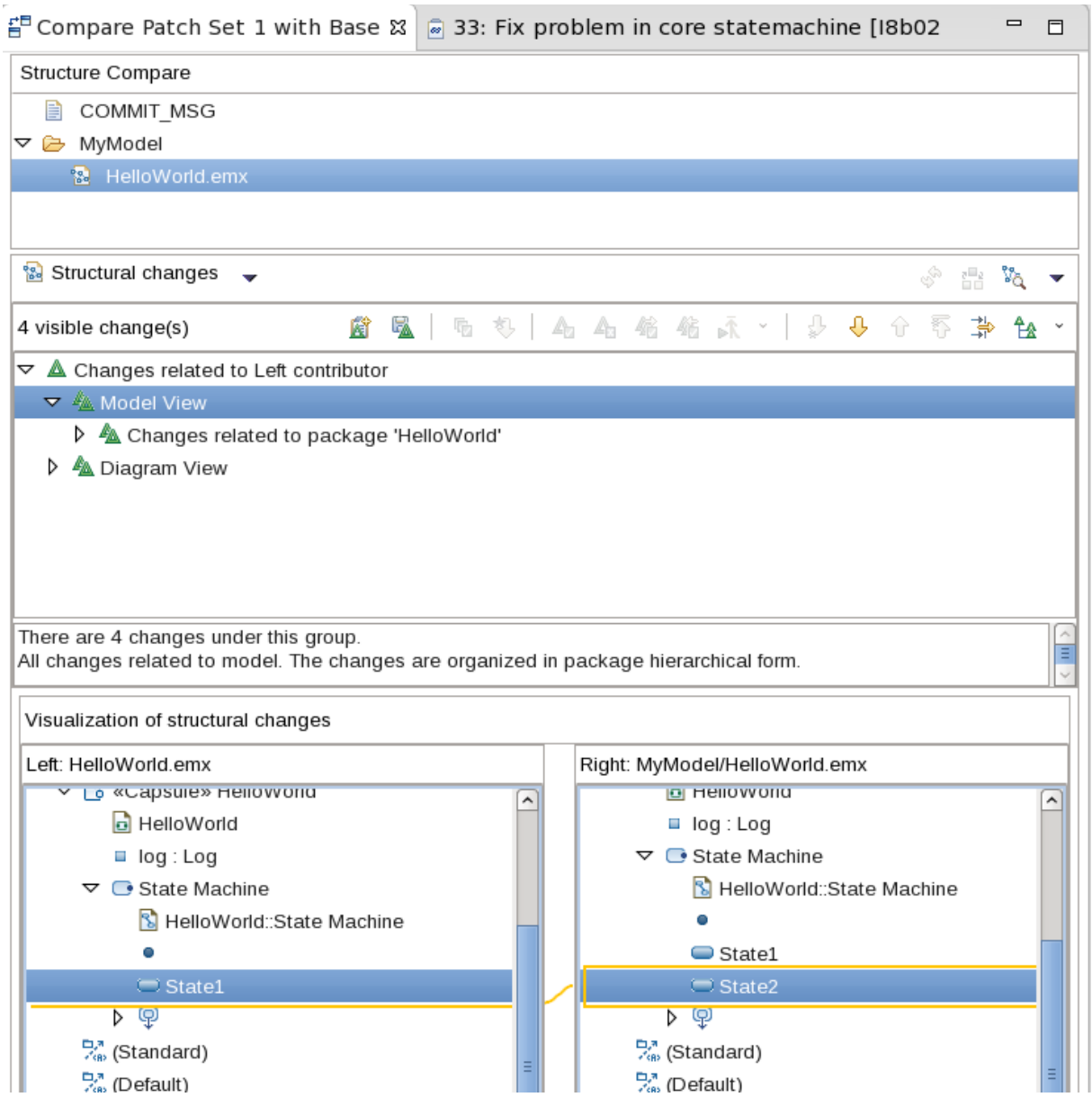
When double-clicking on one of the matches you will get the Mylyn view that contains all relevant information about the review request.



From RSARTE point of view the most interesting possibility is that you can directly access the contents of the change set. If you click on the ”Compare with Base” button you will get a structure compare view that shows all changed files.



Double-clicking one of the model files will show you the differences in the RSARTE compare view.



You can now add comments to the review and take any action that is necessary and the information will be pushed back to the Gerrit review server exactly as if you would have accessed it from the web interface.

Using Amend Commit to Fix Review Comments

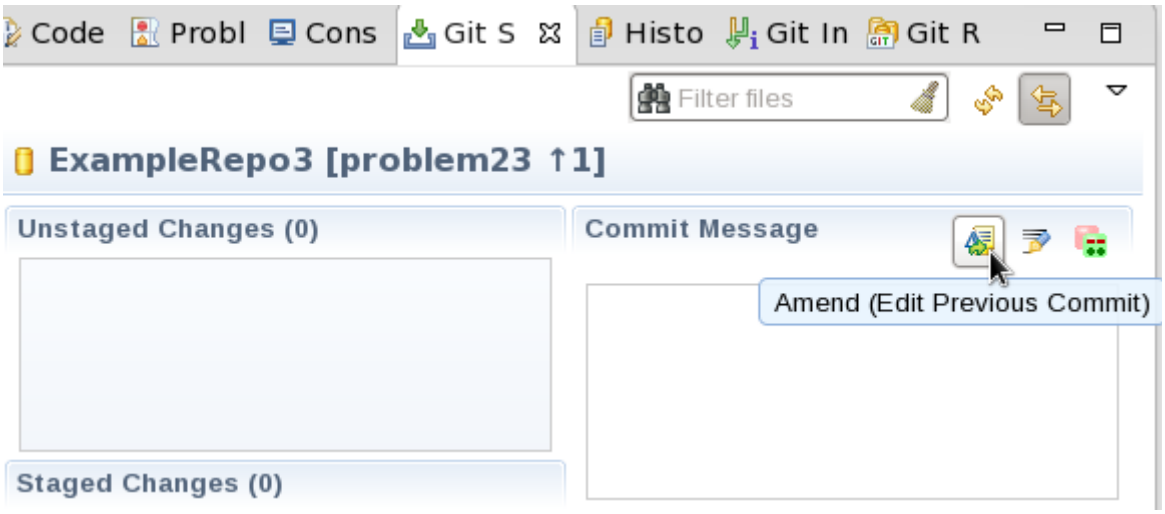
If you have pushed a commit for review to Gerrit, and the reviewer had comments that you would like to fix, then the Git support for amending commits is very handy.

To amend a commit in Git essentially means to make edits to the changes that already have been committed. So it gives you a possibility to re-do some changes.

This can be done using the RSARTE user interface the following way:

- Checkout the commit that you want to change

- Open the Git Staging view
- Click on the ”Amend Commit” button



You now have freedom to make any modifications you would like to fix the review comments. When done stage the changes as usual.

You can now push the commit to Gerrit again as described in Push for Review.
Remember that to make this work in a Gerrit context it is important that you do not change the Change-Id line in the commit comments. You want to make sure that the Change-Id is left as it was to ensure that your fixes will be added to the same review request in Git, instead of creating a new review request.

Other Common Scenarios – Stashing, Resetting and Tagging

In this section we'll go through a few more common actions you likely will need to know about: Stashing, resetting and tagging

Stashing – Temporarily Saving Your Changes

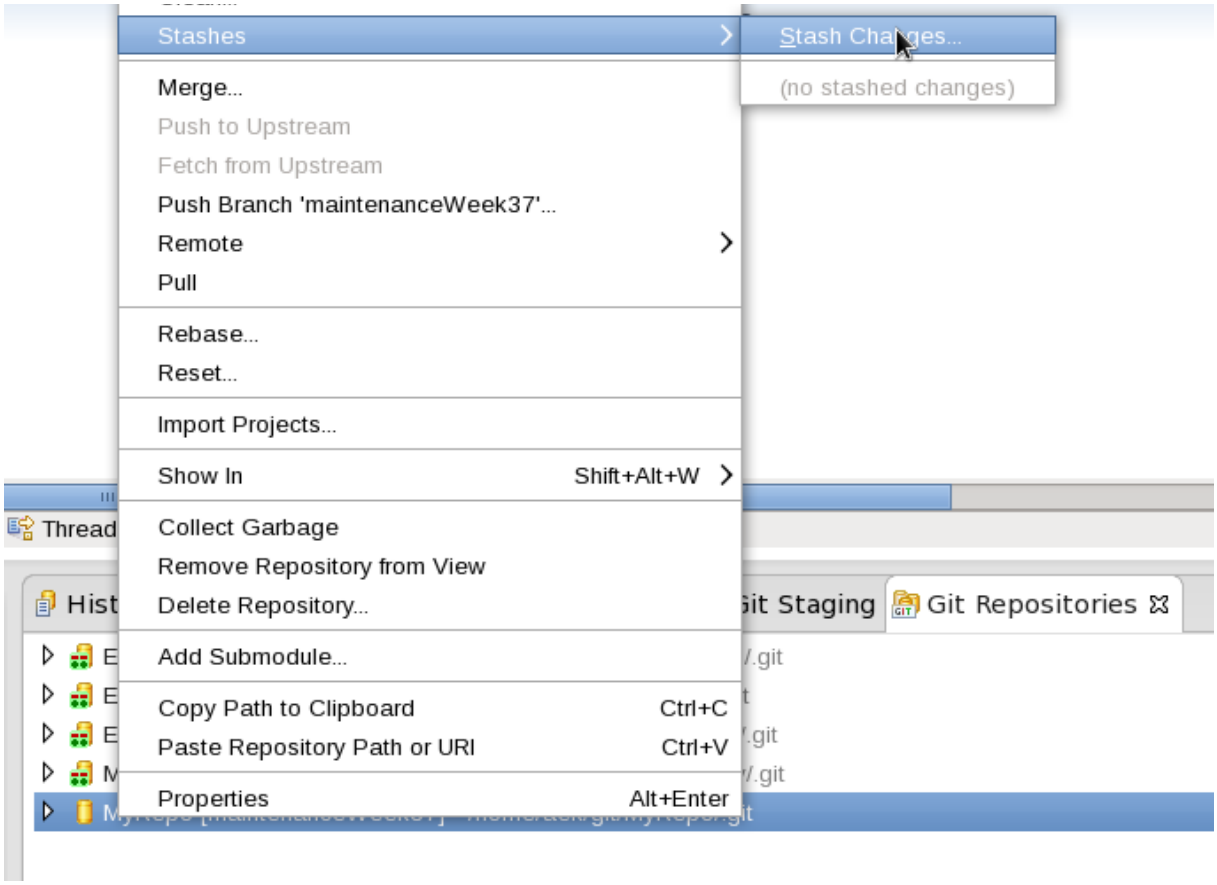
The purpose of the Git stashing feature is to make it possible to save changes you've done in your workspace but that you for some reason don't want to commit to the repository at this time. The typical example is where you are half-way through a change and don't want to commit it when you get some more urgent task you need to handle. Stashing enables you to store them in a safe place and get them back later.

The mechanism thus has two parts:

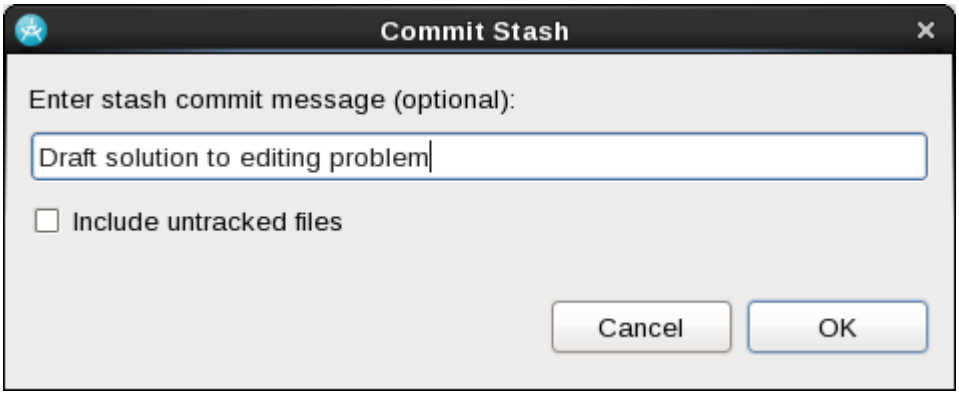
- Save your changes, ”stash” them
- Get your changes back, ”unstash” them

For command line handling of stashes and more information about stashing see for example https://git-scm.com/book/en/v1/Git-Tools-Stashing. In this section we'll focus on the RSARTE user interface actions.

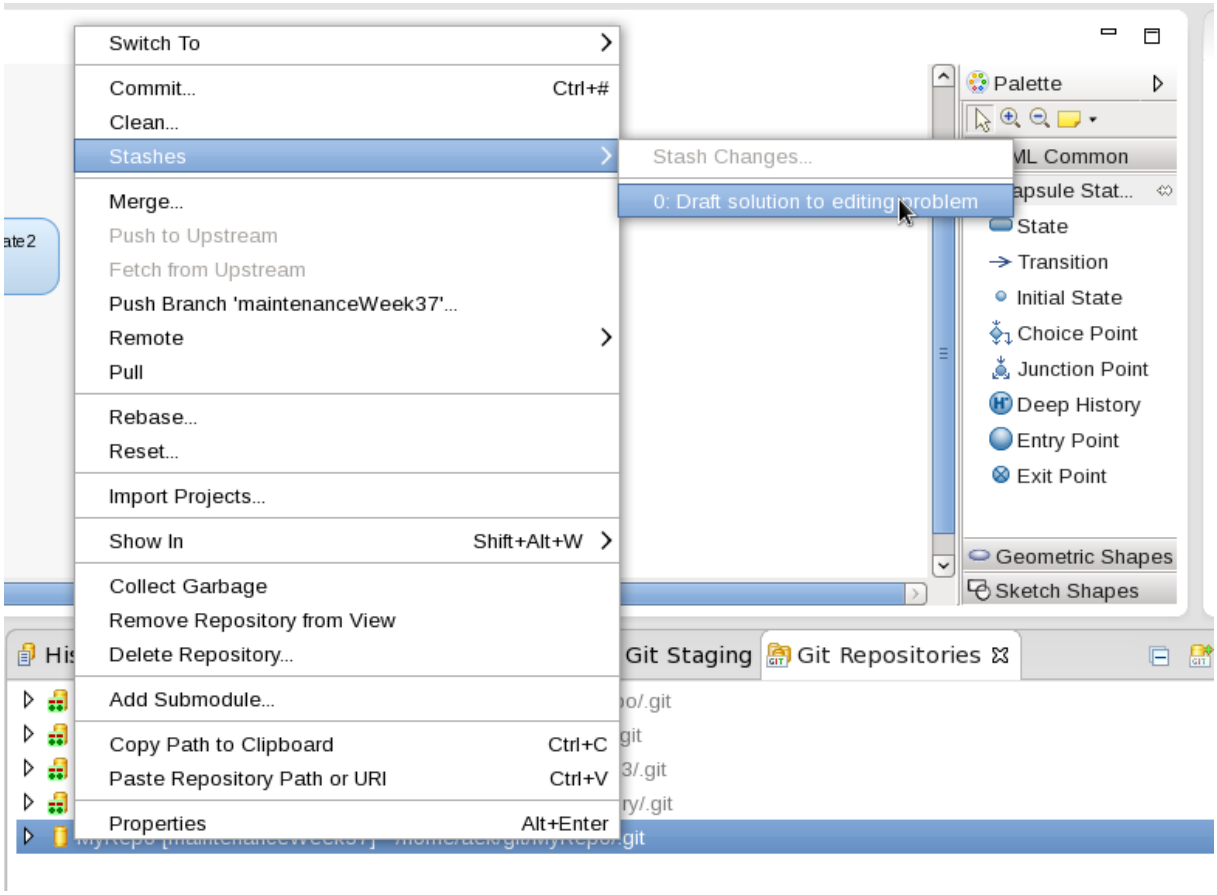
To stash your changes you can use the ”Stashes → Stash Changes” command that is available for a repository in the Git Repositories view. This will save both the modified files in your workspace and the files you have added to the Git index.



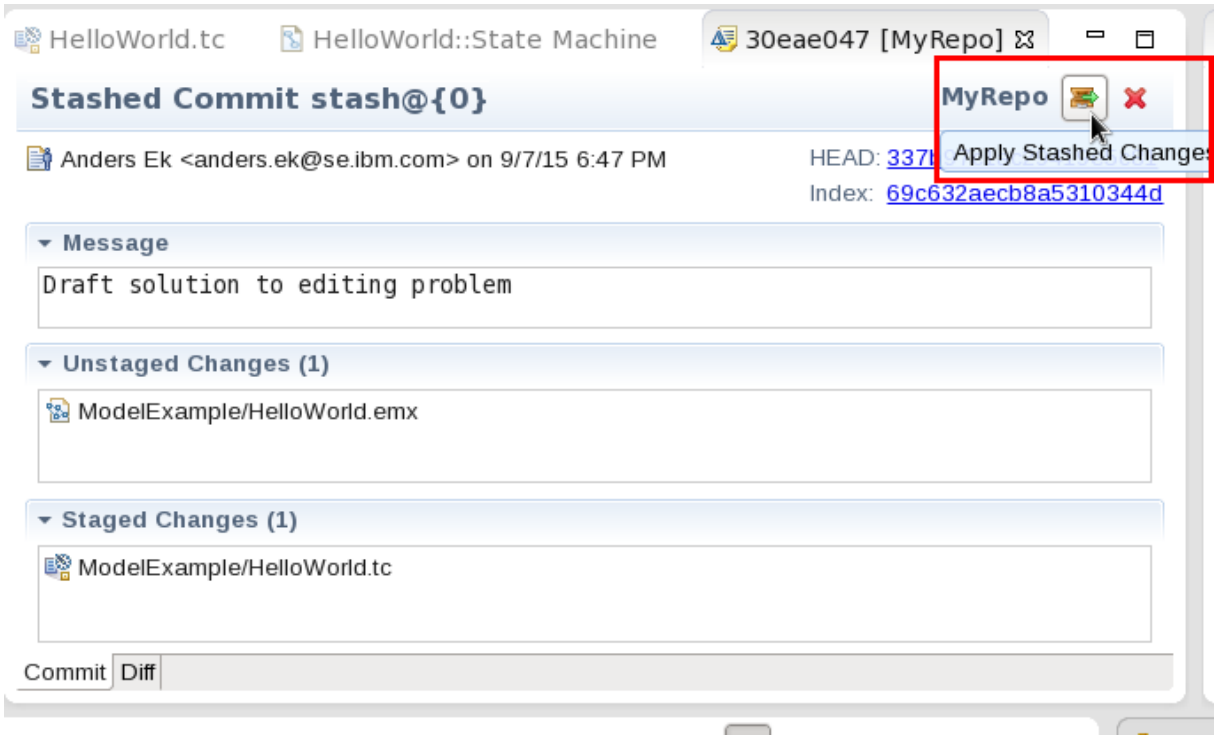
The command will allow you to add a short description of you changes before storing them.



When you are done with the disruption you can get your changes back from the same menu, now selecting the change set you stashed.

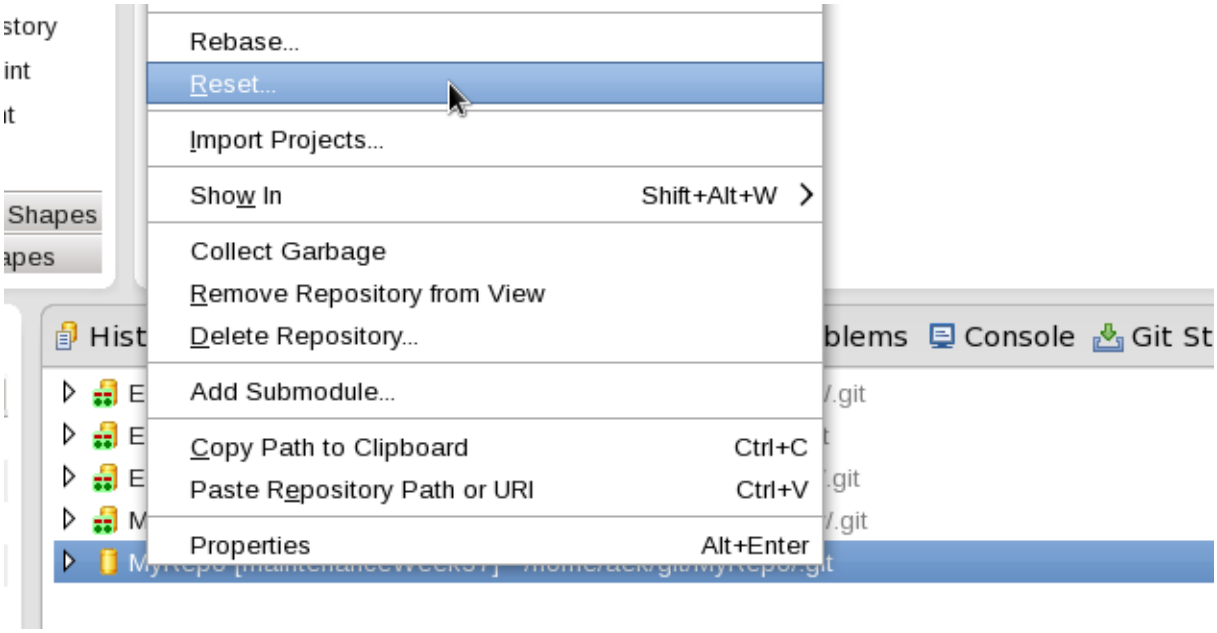


The command will show the stash information view. To actually get back your changes you also need to press the ”Apply Stashed Changes” button.

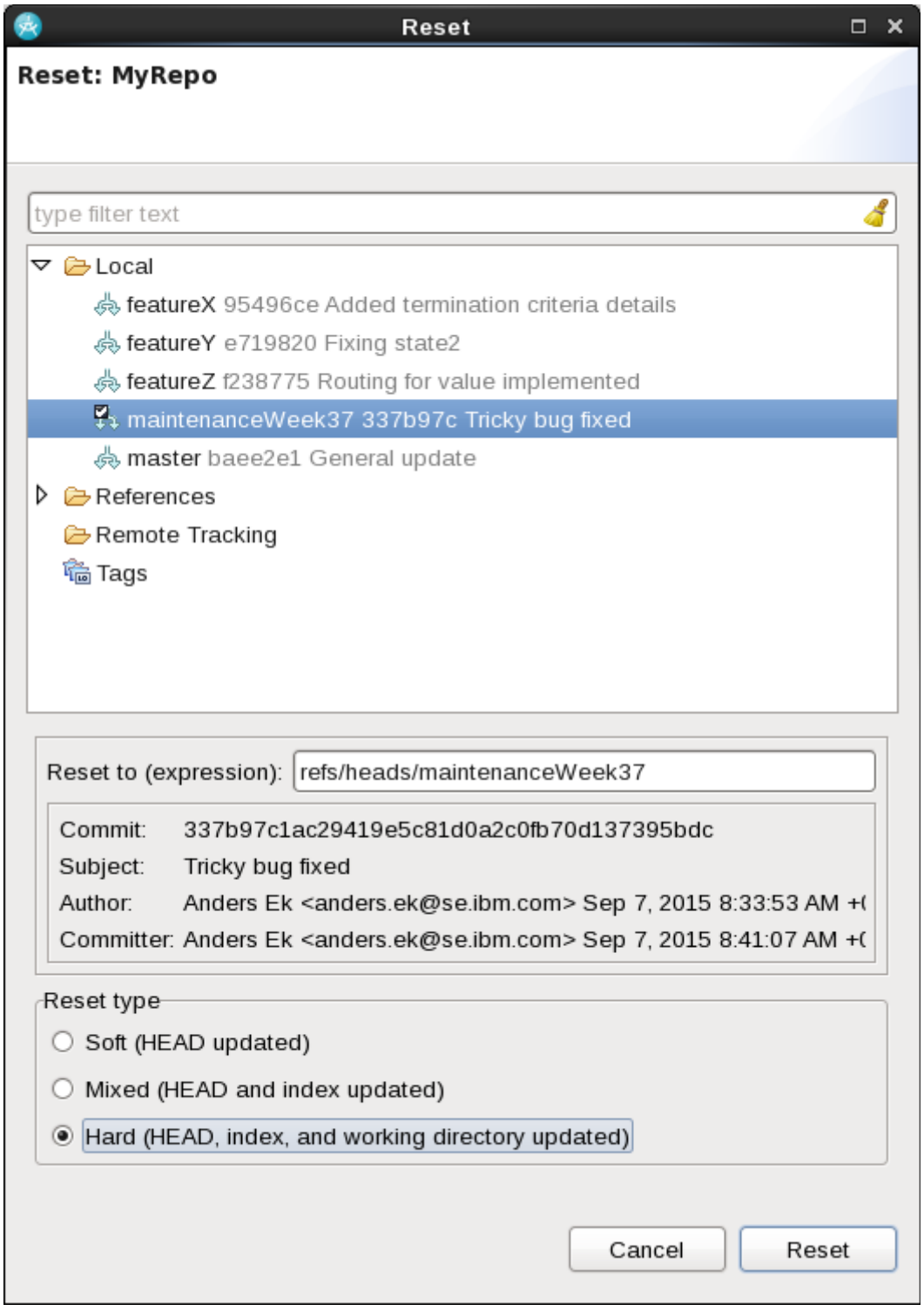


Reset – Reverting Your Changes

If you conclude that the current work you have in progress is all wrong and you want to revert to the last stable version Git provides the ”reset” command available for a repository in the Git Repositories view.



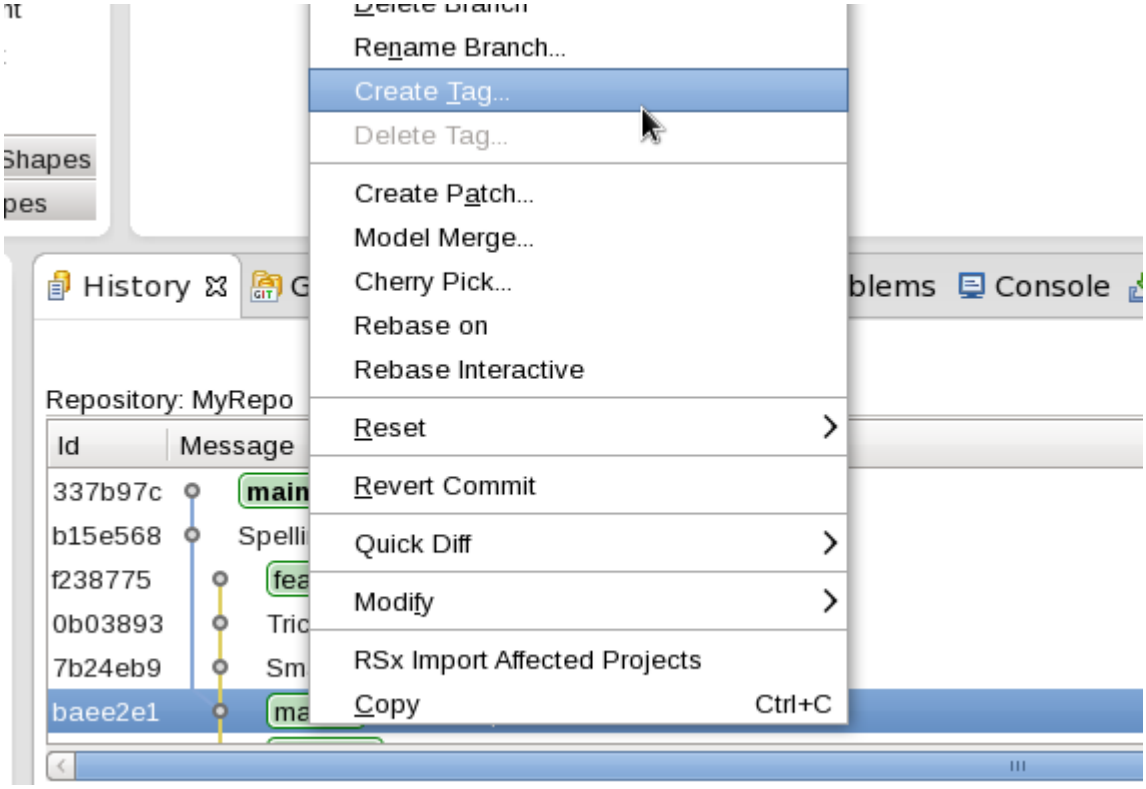
This command allows three different kinds of reset: Soft, mixed and hard. To clean your workspace and start all over the one you want is hard. Soft and mixed allow more complex operations, see for example http://git-scm.com/docs/git-reset for details.



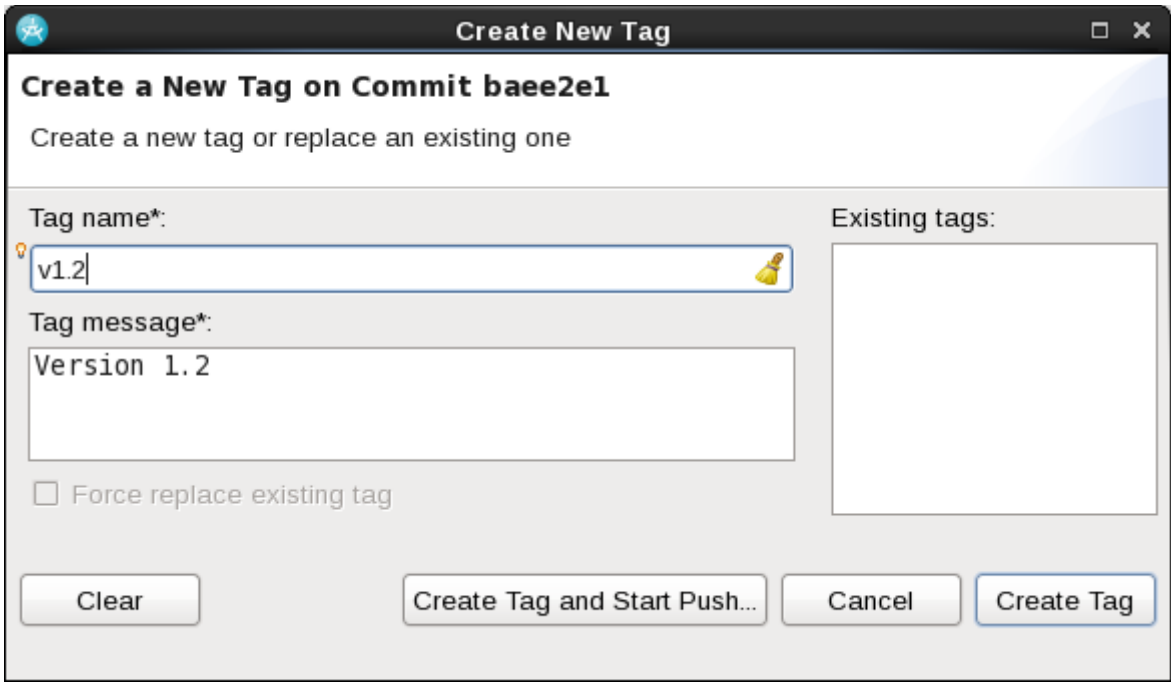
Tagging – Marking Important Commits

A tag is in Git used to put a name on a specific commit. It is typically used to mark for example a release or other external delivery.

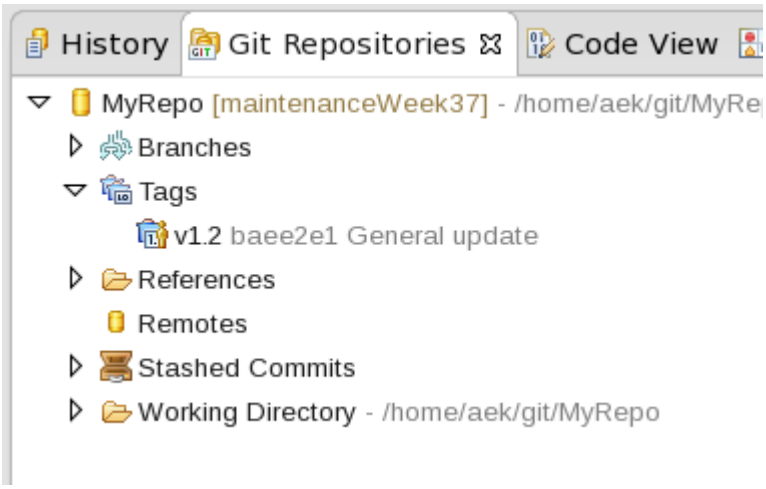
You can create a tag for any commit in the History view using the ”Create Tag” command.



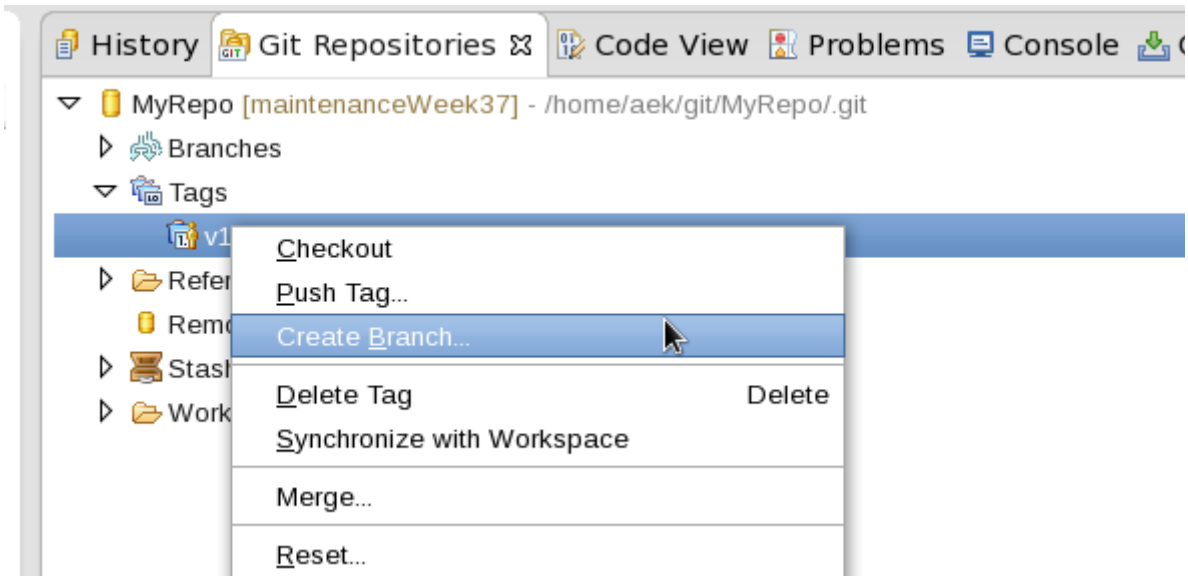
This command will make it possible to specify the tag name and give some additional information about the tag,



If you want to check what tags are defined this information can easily be found in the Git Repositories view, in the Tags folder.



This view is also convenient if you want to create a new branch based on a tag.



Integrating RSARTE with Command Line Git

In this section we will look at what is necessary in order to make command line Git commands that include a merge operation to work for RSARTE models. Note that this is a necessary precondition to safely use RSARTE models from Git command line. Without the merge integration described below Git merge commands may trigger a default text based merge that easily can create corrupt models.

We will also cover some other aspects of integrating RSARTE with a Git customization using hooks.

Command Line RSARTE Invocation

The integration of RSARTE into Git command line handling is primarily done using the RSARTE command line tool for compare/merge. This is implemented by a java application

called cmcmdline.jar that is available in the RSARTE installation in a folder similar to <install-dir>/plugins/com.ibm.xtools.comparemerge.team_7.60.100.v20150603_0908/utm (on a Linux installation).
The details of the command line application is given in section ”Command Line Tool for Compare/Merge” in ”Comparing and Merging Models”. See also the reference guides for the command line tools available from RSARTE Compare/Merge wiki page. In this document we will only cover what is needed for Git integration.

Merge Integration

Git allows the definition of ”merge drivers” in order to make it possible to integrate tools with a special-purpose storage format (like RSARTE) with the Git commands. A description on how this is done can be found in http://git-scm.com/docs/gitattributes. See in particular the section ”Defining a custom merge driver”.

Essentially the following is needed:

- A definition of the merge drivers in the Git configuration (e.g. in .git/config in your repository)
- A mapping of the RSARTE file extensions to the defined drivers using Git attributes (e.g. in the .git/info/attributes file in your repository)

Let us look at an example config definition defining merge for the three types of files handled by RSARTE (.emx, .emf and .tc). As you can see the only difference between the drivers is that the extension is passed to the command line tool using the ”-ext” argument for emx/efx files . This is shown in bold below. The installation specific details that are likely to require modification are also shown in bold.

```
[merge "rsarteemx"]
       name = RSARTE EMX merge driver
       driver = java -cp
/path/to/installation/plugins/com.ibm.xtools.comparemerge.te
am_7.60.100.v20150603_0908/utm/cmcmdline.jar
com.ibm.xtools.comparemerge.cmcmdline.CMTool merge -ext=emx
-ancestor %O -left %A -right %B -out %A
[merge "rsarteefx"]
       name = RSARTE EFX merge driver
       driver = java -cp
/path/to/installation/plugins/com.ibm.xtools.comparemerge.te
am_7.60.100.v20150603_0908/utm/cmcmdline.jar
com.ibm.xtools.comparemerge.cmcmdline.CMTool merge -ext=efx
-ancestor %O -left %A -right %B -out %A
[merge "rsartetc"]
       name = RSARTE EFX merge driver
       driver = java -cp
/path/to/installation/plugins/com.ibm.xtools.comparemerge.te
am_7.60.100.v20150603_0908/utm/cmcmdline.jar
com.ibm.xtools.comparemerge.cmcmdline.CMTool merge -ancestor
%O -left %A -right %B -out %A
```

An example definition from an Git attributes file:

```
*.emx merge=rsarteemx
*.efx merge=rsarteefx
*.tc merge=rsartetc
```

When these (or similar) definitions are done all command line Git commands that trigger a merge Invocation will correctly handle RSARTE model files. Note however that this will only trigger a file-by-file merge. To invoke a logical model merge or a closure merge either use the RSARTE graphical user interface or call the command line application cmcmdline.jar directly as is describe in section ”Command Line Tool for Compare/Merge” in ”Comparing and Merging Models”.

Also note that this integration only handles ”merge”, not the conflict resolution that requires a ”merge tool” integration. Currently we recommend the usage of the graphical user interface in RSARTE to handle conflict resolution.

Git Hooks

Git allows various hooks to be associated with various actions. In general this works fine together with RSARTE, but there is one exception: Scripts that require user interaction, for example asking for confirmation of some action, will NOT work from the RSARTE user interface. From a users point of view it will appear as if the tool hangs when this happens. A typically example is a pre-commit hook that asks for user input. This will be triggered e.g. when pressing the ”Commit” button in the Git Staging View and the tool will appear to hang.